

Outsource Your Software Development? Insource Your Quality!

Making the decision to outsource the development of your critical application is not easy. There are pros and cons, but, once the decision is made, it is imperative that the **quality you paid for and expected is the quality you receive.**

Ask yourself these questions:

1. What is the quality of the code that has been developed?
2. Is it too complex, making it unmaintainable and unreliable?
3. How thoroughly was it tested prior to delivery back to you?
4. Are you convinced that the most complex areas of your critical application are being tested?
5. Is the code quality and test coverage trending in a positive direction?

If you do outsource, you owe it to yourself, your organization, and most importantly your customers to know the answers to these questions.

McCabe IQ provides the answers using advanced static and dynamic analysis technology to help you focus your attention on the most complex and risky areas of your code base.

McCabe IQ's breakthrough visualization techniques, enterprise reporting engine, and executive dashboard provide you with a complete picture of what is being produced.

It's time you made the most of your outsourcing investment and benefited from our 30+ years of software quality research and development.

Don't just think your code is good.
Be sure of it, with McCabe IQ.

Download "Code Quality Metrics in Management of Outsourced Development" at www.mccabe.com/bettersoftware/digital.htm.



McCabe IQ
Your Software...Better™

McCabe
SOFTWARE

www.mccabe.com / 800-638-6316



May/June 2009

\$9.95 www.StickyMinds.com

BETTER SOFTWARE

The Print Companion to StickyMinds.com

STAND AND DELIVER
Panic-free presentations

THE GOOD, THE BAD, AND THE VIRTUAL LAB
Is VLA right for you?



**CERTIFIED
SOFTWARE
TESTER**



eLearning

CLICK HERE

**FIND THE BUG INSIDE & WIN
AN AMAZON GIFT CARD!**

Cut Dev Costs... Without Cutting Heads.



*It's tough to look
at your team and
decide who should stay
and who should go.*

Rally's customers are proven to be
50% faster to market and
25% more productive,
so you can avoid cutting heads.
Plus Rally is the only Application Lifecycle
Management provider to ensure
your Agile success with
a money-back guarantee.

**Learn more about our Guaranteed Success Program at
www.rallydev.com**



Scaling Software Agility

© 2009 Rally Software Development Corp

Save time while protecting software quality.



© 2009 Seapine Software, Inc. All rights reserved.

Swiftcover.com cut their testing time in half with TestTrack Studio and QA Wizard Pro, while still providing the quality their customers expect.

Seapine's end-to-end Software Quality Assurance (SQA) solutions help you deliver quality products faster. Start with **QA Wizard Pro** for automated testing and add **TestTrack Studio** for issue tracking and test case management—integrated quality assurance solutions that together reduce testing time, saving you money and improving customer satisfaction.

- Reduce quality assurance costs with automated functional and regression testing.
- Manage test case development, defect assignments, and QA verification with one application.
- Track which test cases have been automated, schedule script runs, and link test results with defects.

“*So much of success boils down to time. QA Wizard Pro and TestTrack Studio allow us to be more profitable because we do more in less time.* — Test Manager, Swiftcover”

Learn how to test faster while protecting quality. Visit **www.seapine.com/betterswift**

Test your reality with Cognizant. Compete with confidence.

Whether it's supplementing your existing program or providing an end-to-end solution, whether it's a long term testing roadmap or detailed process improvements, you can count on Cognizant's consistency, discipline, and operational efficiency.

There are many advantages to working with Cognizant. We'll help you look beyond the obvious, provide you with independent suggestions for quality improvements that will mitigate risks and improve software performance.

With Cognizant, you get certified continuous business readiness, lower costs, and accelerated results unmatched in the industry.



Cognizant | **Testing Services**
Passion for building stronger businesses

World Headquarters

Cognizant Technology Solutions
500 Frank W. Burr Boulevard
Teaneck, NJ 07666
Ph: +1 201 801 0233
Fax: +1 201 801 0243
Toll free: +1 888 937 3277
Email: inquiry@cognizant.com
Website: www.cognizant.com/testing

Cover Story

SCRUM 20

For organizations trying to do more with less in the current economy, knowing where to turn for help can be a big question mark. Scrum is one possible solution that brings teams together through frequent communication and high-impact collaboration, resulting in increased productivity and an ability to build a better product faster. *by Laszlo Szalvay*

Features



Columns & Departments

In Every Issue

Editor's Note **4**

Contributors **6**

10 Things You Might
Not Know About ... **38**

Ad Index **40**

Better Software magazine—The print companion to StickyMinds.com brings you the hands-on, knowledge-building information you need to run smarter projects and deliver better products that win in the marketplace and positively affect the bottom line. **Subscribe today to get six issues.**

Visit www.BetterSoftware.com
or call 800.450.7854.



VIRTUAL REALITIES 26

Virtual lab automation (VLA) promises quantifiable benefits for application development and test organizations. This article discusses best practices and common pitfalls associated with adopting a VLA solution, outlines the steps to evaluate a virtualization solution for your test organization, and provides resources to help you get started. *by Ian Knox*

WHAT TO EXPECT WHEN YOU'RE AUTOMATING TESTING 32

Writing automated tests for your existing codebase can be a daunting challenge. Learn strategies to kick-start your testing and some solutions to problems teams frequently encounter. *by Daniel Wellman*

TECHNICALLY SPEAKING 9

Time to Let Go of Obsolete Jobs • *by Lee Copeland*

When new paradigms are created, new jobs are often created with them. And sometimes, existing jobs are no longer relevant.

CODE CRAFT 10

GUT Instinct • *by Kevlin Henney*

To be sustainable, the style of a unit test is just as important as the style of any other code. Perhaps a little surprisingly, the most commonly favored test partitioning style does not meet these expectations.

TEST CONNECTION 14

Issues about Metrics about Bugs • *by Michael Bolton*

Managers often use metrics to help make decisions about the state of the product or the quality of the work done by the test group. Yet measurements derived from bug counts can be highly misleading because a "bug" isn't a tangible, countable thing.

MANAGEMENT CHRONICLES 16

Crash Course in Proficient Presenting • *by Naomi Karten*

The support of an experienced speaker and coach who offers advice and encouragement can help you become a proficient, panic-free presenter.

THE LAST WORD 39

Putting the Kart before the Horse? • *by Antony Marcano*

A former go-kart racer recounts a racing experience that helps him explain what goes wrong for many organizations adopting Scrum as a first attempt to "go agile."

StickyMinds.com We invite you to visit StickyMinds.com, the online companion to *Better Software* magazine. StickyMinds.com covers the same pertinent topics as the magazine, putting the power of information at the click of your mouse. Weekly columns, headline-making bugs, hundreds of technical papers, an online tools guide, discussion boards, and so much more make StickyMinds.com your site for 24/7 brainfood to help you build better software.

Editor's Note

THAT'S EASY FOR YOU TO SAY



I recently presented some research to Lee Copeland, *Better Software* magazine managing technical editor and Software Quality Engineering conference chair. It didn't go well. I rambled and stuttered and ummed and pretty much did everything I could *not* to convey the information clearly and concisely. Fortunately, Lee and I have a good working relationship, so he didn't hold it against me, but I don't think he'll be offering me a keynote spot at a conference any time soon.

If I had been asked to prepare a document outlining my research, I would have had no problem stringing words together to form sentences that flow and make perfect sense. But something about opening my mouth and verbally conveying a thought turns me into a raving lunatic.

I don't know many people who truly enjoy public speaking, but the ability to verbally disseminate information to others in a way that is easy to understand and digest is a pretty important skill and one that every professional should possess. I apparently need some help in this area and, in fact, should have taken some tips from this issue's Management Chronicles, "Crash Course in Proficient Presenting." The author, Naomi Karten, is an accomplished speaker and seminar leader. In her article she describes a situation to which many of us can relate. A worker is volunteered by his manager to present some valuable lessons learned on a completed project. The worker is uncomfortable in this role and turns to an experienced presenter for advice. Naomi's article has several useful tips including how to harness nervousness to energize your presentation, how to project confidence to your audience, and the best way to get your nerves under control.

Naomi has been sharing her knowledge with *Better Software* magazine and StickyMinds.com readers for many years as a Management Chronicles contributor and columnist, and now we have the benefit of her insight on a much more regular basis. Naomi has joined our growing roster of bloggers who are sharing their experience and expertise on StickyMinds.com. If you haven't had a chance to visit our blogs, go to blogs.stickyminds.com and add us to your RSS feed.

As always, I hope you enjoy this issue of *Better Software* magazine—and our blogs! Drop me a note to let me know how you've put this issue to work for you.

Happy Reading!

Heather Shanholtzer
HShanholtzer@sqa.com

BETTER SOFTWARE

Publisher
Wayne Middleton

Vice President of Publishing
Holly N. Bourquin

Editor in Chief
Heather Shanholtzer

Editorial

Managing Technical Editor
Lee Copeland

Editor, StickyMinds.com
Francesca Matteu

Managing Editor, Multimedia
Joseph McAllister

Production Coordinator
Cheryl M. Burke

Design

Creative Director
Catherine J. Clinger

Advertising

Senior Advertising Sales Manager
Shae Young

Production Coordinator
April Evans

Circulation and Marketing

Circulation Coordinator
Jamie Green-Gago

Marketing Coordinator
Sidney White

A PUBLICATION OF
SOFTWARE QUALITY ENGINEERING



CONTACT US

Editors: editors@bettersoftware.com

Subscriber Services: info@better-software.com

Phone: 904.278.0524, 888.268.8770

Fax: 904.278.4380

Address:

Better Software magazine
Software Quality Engineering, Inc.
330 Corporate Way, Suite 300
Orange Park, FL 32073

**The de facto standard for quality management
for game developers around the globe.**



Don't you think it's time you find out why?



DevTest Studio

The integrated solution for defect tracking, test management and automated testing

DevTest

Use DevTest to manage your testing

- Create a central repository for your test cases, knowledge items and automation scripts
- Schedule releases and test cycles using a wizard-driven interface
- Execute test assignments and submit defects from the same interface
- Track results with real-time dashboards and reports

DevTrack

Use DevTrack to track defects/issues

- Track each issue through a definable workflow
- SCM integration – track fixes against their source code deliverables
- Deploy a resolution across multiple releases, versions and products
- Reporting and metrics to illustrate the entire defect lifecycle

TestLink

Use TestLink to automate your testing

- Add automated tests to the DevTest test library
- Schedule automated tests along with manual tests
- Launch automated tests from the DevTest interface
- Track automation results with real-time dashboards and reports

TechExcel


www.techexcel.com | 1-800-439-7782

Contributors



MICHAEL BOLTON lives in Toronto and teaches heuristics and exploratory testing in Canada, the United States, and other countries. He is co-author, with James Bach, of *Rapid Software Testing* and a regular contributor to *Better Software* magazine. Contact Michael at mb@developsense.com.



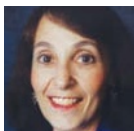
LEE COPELAND has more than thirty years of experience in the field of software development and testing. He has worked as a programmer, development director, process improvement leader, and consultant. Based on his experience, Lee has developed and taught a number of training courses focusing on software testing and development issues. Lee is the managing technical editor for *Better Software* magazine, a regular columnist for StickyMinds.com, and the author of *A Practitioner's Guide to Software Test Design*. Contact Lee at lcopeland@sqe.com. 



RICK CRAIG has a wide range of experiences as a tester and test manager. Currently a consultant with SQE Training, Rick specializes in metrics, management, and process improvement. He has spoken at conferences around the world since 1984, including every STAR conference. Rick is the co-author of *Systematic Software Testing* and is a colonel in the USMC Reserve.




KEVLIN HENNEY is an independent consultant and trainer based in the UK. He provides consultancy and training in programming techniques, software architecture, and development process. Kevlin is co-author of two recent books on patterns, *A Pattern Language for Distributed Computing* and *On Patterns and Pattern Languages*.



NAOMI KARTEN is a highly experienced speaker and seminar leader who draws from her psychology and IT backgrounds to help organizations improve customer satisfaction, manage change, and strengthen teamwork. She has delivered seminars and keynotes to more than 100,000 people internationally. Naomi's newest book is *Changing How You Manage and Communicate Change*. Her other books include *Managing Expectations*, *How to Establish Service Level Agreements*, and *How to Survive, Excel and Advance as an Introvert*. She is a regular columnist for StickyMinds.com. When not working, Naomi's passion is skiing deep powder. Contact her at naomi@nkarten.com or www.nkarten.com.



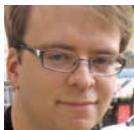
As director of product management, **IAN KNOX** is responsible for all aspects of Skytap's product management and go-to-market strategy. Ian joined Skytap from Microsoft Corporation, where he was group product manager for Microsoft Visual Studio. Prior to Microsoft, Ian was a principal consultant at PricewaterhouseCoopers, where he worked on global software delivery projects for Fortune 500 clients. 



ANTONY MARCANO has fourteen years of experience as a tester and developer, the last nine of which have been primarily on agile projects. Now, as an independent coach and consultant, he helps teams realize the benefits associated with agile software development. Antony, a former *Better Software* magazine technical editor, is also creator and curator of the popular Web site testingReflections.com and co-creator of PairWith.Us, a live, interactive pair-programming Web-cast. Antony is referenced in several books on agile practices and is a regular guest lecturer on test-driven development and acceptance test-driven development at Oxford University. Contact him at [antony.marciano@testing, Reflections.com](mailto:antony.marciano@testing,Reflections.com) or follow him on Twitter at twitter.com/AntonyMarcano.



As cofounder and president of Danube Technologies, Inc., **LASZLO SZALVAY** drives the company's vision, creating business initiatives that influence everything from sales and marketing to human resources, employee retention, and accounting. Laszlo lives in Portland, OR with his wife, Alison, and their daughter, Claire. In his free time, Laszlo enjoys barbecuing, collecting Oregon Pinot noir, playing speed chess, and spending time with old friends and family.



DANIEL WELLMAN is a technical lead at Cyrus Innovation, a leading New York agile consultancy, where he leads development projects and coaches teams on adopting agile software development practices. With more than ten years of experience building software systems, Daniel is an expert in agile methodologies, object-oriented design, and test-driven development. Contact Daniel at dan@danielwellman.com.

BETTER SOFTWARE

CONFERENCE & EXPO

JUNE 8-12, 2009
LAS VEGAS, NEVADA
THE VENETIAN

*There's Still Time!
Reserve Your Seat Now.*

Conference Sponsor:



www.sqe.com/bsce



KEYNOTES BY INTERNATIONAL EXPERTS



Tim Lister
Atlantic Systems Guild, Inc.



Andy Kaufman
*Institute for Leadership
Excellence & Development, Inc.*



Michele Sliger
Sliger Consulting, Inc.



Jonathan Kohl
Kohl Concepts, Inc.



NEW FOR 2009!
a co-located event

Agile Leadership Summit
Friday, June 12, 2009



Industry Sponsors:



In Cooperation With:



Media Sponsors:



Methods & Tools



Outsource Your Software Development? **Insource Your Quality.**

Making the decision to outsource the development of your critical application is not easy. There are pros and cons, but, once the decision is made, it is imperative that the **quality you paid for** and expected is the **quality you receive**.

Ask yourself this:

What is the quality of the code that has been developed?

Is it too complex, making it unmaintainable and unreliable?

How thoroughly was it tested prior to delivery?

Are you convinced that the most complex areas of your critical application were tested?

Is the code quality and test coverage trending in a positive direction?

If you do outsource, you owe it to yourself, your organization, and most importantly your customers to know the answers to these questions.

McCabe IQ provides those answers using advanced static and dynamic analysis technology to help you focus your attention on the most complex and risky areas of your code base.

McCabe IQ's breakthrough visualization techniques, enterprise reporting engine, and executive dashboard provide you with a complete picture of what is being produced.

It's time you made the most of your outsourcing investment and benefitted from over 30 years of software quality research and development.

Don't just think your code is good. Be sure of it, with McCabe IQ.

McCabe
SOFTWARE

Download "Code Quality Metrics in Management of Outsourced Development" at www.mccabe.com/bettersoftware, or call 800-638-6316 to learn more.



McCabe **IQ**
Your Software...Better™

Is your **agile fragile**?

Unit test faster, better, and cheaper with AgitarOne

AgitarOne helps you work faster, better, and more cost efficiently as you develop and maintain your Java applications.

'AgitarOne JUnit Generator' is the fastest and easiest way to create a thorough suite of JUnit tests, both for new code and for legacy applications. It helps you find regressions and makes it safer and easier to improve your code and reduce the cost to maintain it.

AgitarOne JUnit Generator can generate 250,000 lines or more of JUnit per hour and routinely achieves JUnit coverage of 80% or better.

**Get 80%
- or better -
code coverage
at the push
of a button**

'AgitarOne Agitator' helps developers understand the behavior of their code as they write it. This helps you prevent bugs and eliminate the code complexity that becomes tomorrow's maintenance headache.

Based on our breakthrough innovation called "software agitation", AgitarOne Agitator helps developers not only validate the expected behavior of their code but also discover unexpected behavior.

AgitarOne helps you unit test better and faster, and is the best way to create, use, and manage the unit tests needed to be truly agile.

Request your evaluation now and get started at www.agitar.com/bettersoftware.

Agitar
TECHNOLOGIES

Time to Let Go of Obsolete Jobs

by Lee Copeland

Town crier, elevator operator, gas lamp lighter, telegraph operator—you probably haven't seen many help wanted ads for these occupations lately. Why? Because these occupations are gone—obsolete, unnecessary, outdated. We just don't need them any more. When new paradigms are created, new jobs are often created with them. And sometimes, existing jobs are no longer relevant.

In the agile world, teams are encouraged to be cross-functional, self-organizing, and composed without consideration of the corporate hierarchy. Gone are the command-and-control managers responsible for planning, organizing, leading, controlling, monitoring, and motivating. Rather, the team is responsible for those things now. Self-organizing teams negotiate commitments with their customers and executive management, make plans, do the work, track their progress, assess project progress and risks, report status and progress, focus on quality outcomes rather than delivery for delivery's sake, and fluidly take on tasks and responsibilities to move the work forward.

Recently, I attended the annual Software Quality Association of Denver's (SQuAD) Software Quality Conference. SQuAD is a nonprofit organization that supports quality assurance professionals in the Mile High City and Denver metro area. The conference invites internationally known testing experts to share their knowledge and experience.

Lisa Crispin presented a tutorial, "The Tester Who Came in from the Cold—Helping Testers Transition to Agile," based on her experience leading agile teams. As she described the integration of developers and testers to form self-organizing teams, Lisa posed the question, "What does the QA manager do now?"

While self-organizing teams assume the responsibility for "getting it done," they often need assistance from outside the team. Types of assistance may include:

- **Champion**—All teams need a champion to inspire them with a vision, support them with resources, cheer their successes, encourage them when times are difficult, and defend them against internal and external attacks.
- **Observer/Evaluator**—Even the most effective team can benefit from feedback from an outside observer. It's difficult to step outside to observe what is working well and what is not. The team's retrospective process provides a "first opinion" of how things are progressing. An external observer can provide a complementary "second opinion."
- **Gatekeeper**—Teams are constantly bombarded from the outside with "good suggestions." The gatekeeper evaluates information and controls its flow into the team. Even the best ideas, impinging on the team too rapidly for evaluation and assimilation, will be disruptive.
- **Mentor**—Even the most effective self-organizing teams don't know everything. The mentor provides skill-improvement experiences for individuals and for the team.
- **Trouble shooter**—Sometimes even the best teams get stuck. In those cases, having a "fixer" to get things unstuck is invaluable. Every organization needs a Winston Wolfe.
- **Administrator**—In most organizations, timecards must be signed, leave scheduled, supplies ordered, and myriad other administrative details handled. The administrator provides all those services for the team.

I don't know what the job title is for the person who performs all this assistance. In actuality, it's probably a number of people, and these services and functions should be formalized in the organization. But I do know what the job is not

called—it is not "QA manager." Within agile development using self-organizing teams, the title of QA manager is now obsolete, unnecessary, and outdated. Rather than fight it and redefine the job with a set of flimsy responsibilities, let's be dignified as it goes "gentle into that good night." (If you are the QA manager, find a job where you can contribute more. There's plenty of important work to be done.)

Now, to go one step further, it's not just jobs that become obsolete—it's entire organizational structures. My former colleague Norm Sammis tells us that the organization chart (the one posted on the wall) is merely a snapshot of how the organization used to be structured. Internally, organizations are constantly evolving in an attempt to become more effective. New informal leaders, team structures, and relationships emerge. When the actual organization becomes too far disconnected from the chart, executive management "reorganizes"—it updates the chart to better reflect reality. Self-organizing teams don't fit well in the classic hierarchical organization chart (think rows of rectangles, arranged like a pyramid, reaching to the heavens—or at least the CEO). Where is the team in this organization chart? It's scattered helter-skelter all over.

Perhaps it's time to re-engineer the organization, placing the self-organizing teams in the center, surrounded by the services and functions that support them (think concentric circles with the team in the center).

This is not a new idea—surgical teams in hospitals have been organized this way for decades. Surgeons are assisted by anesthesiologists, nurses, and technicians. Support services and functions surround them. In *The Mythical Man-Month*, first published in 1975, Fred Brooks proposed a similar organization style for software development teams. It's time to embrace the idea that to make self-organizing teams most effective, the organizational context in which they exist must be "tuned" for better efficiency. **{end}**

GUT Instinct

by Kevlin Henney

In one of my previous columns “Programming with GUTs,” [1] we looked at some of the stylistic qualities that make up *good unit tests*, or GUTs, following Alistair Cockburn’s coinage [2]. A tour of articles, books, and the blogosphere reveals a lot of emphasis on the importance of quality in production code and exhortations to test, but perhaps less on the quality of the tests themselves. Automated unit test cases are also code, and the case for quality applies just as much to their sustainability as to that of production code.

Obviously, GUTs are better than no unit tests (NUTs), but it is not clear that bad unit tests (BUTs) are necessarily better than NUTs. BUTs can hold back both a team and a codebase by restricting rather than supporting the kinds of changes that can be made. Without the appropriate sensibility, automated tests can degenerate into an accumulated jumble of test cases with no obvious narrative or direction.

Procedural Problems

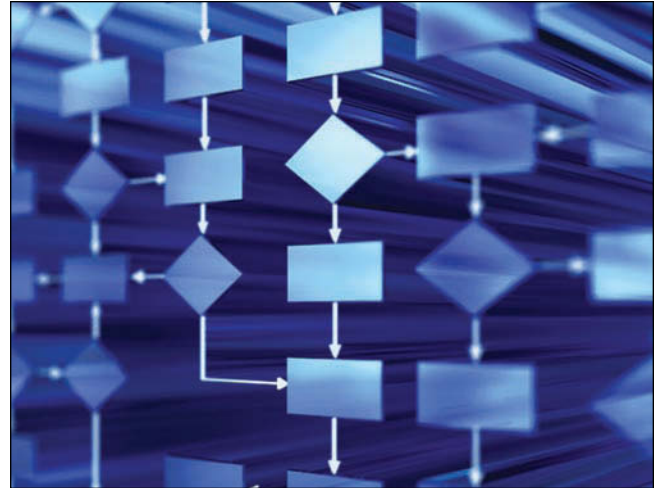
What are unit tests for? One of the most common responses is that unit tests are used to demonstrate that code is doing the “right thing.” But, what exactly *is* the right thing? Without a clear idea of what the right thing is, programmers can end up reducing testing to arbitrary and dispirited code poking—or to nothing at all.

Common but unjustified beliefs about testing are not just restricted to purpose; they can also be found in practice. For example, perhaps the most popular and seemingly intuitive technique for partitioning test cases is in a procedural style: For each method in a class, write a corresponding test method; for method `foo`, define `testFoo`.

Aligning *method under test* with *test method* undermines the simpler and more important correlation of a *test case* corresponding to a *test method*: Testing a method with rich behavior in a single method fails to differentiate the underlying test cases; trying to test query methods on their own misses the point that their results can only be interpreted meaningfully within specific contexts, following previous calls. Lumping together all the common cases, boundary cases, and failure cases for a method into a single test case makes it harder to see the forest for the trees. Such arbitrary grouping is a surefire way to create messy tests and to overlook important cases.

If test cases are intended to define and confirm the behavior of a class, then the test cases need to align with cases of use that may cut across many methods, rather than align with methods, which may enclose a wide range of behavior. If test cases are intended to define and confirm the behavior of a class, then the test cases need to align with cases of use that may cut across many methods, rather than align with methods, which may enclose a wide range of behavior.

The popularity of the procedural testing style is carried



ISTOCKPHOTO

through to automated perfection in many code-generating wizards, which generate test stubs for each method in a production class. These address the trivial work—writing method signatures—and carefully sidestep the hard part—the details of the actual test. Such tools nudge programmers into the wrong frame of mind for describing the usage behavior of their code. Along with the “helpful” tools that autolitter code with setters and getters, such tools bear a stronger resemblance to so(u)rcer’s apprentices than they do to wizards.

From Prodding Procedures to Revealing Requirements

It seems clear that procedural testing does not work well with objects, but is the same also true for code that is procedural or functional in style? Consider, for example, a single function that determines whether or not a given year is a leap year. To keep things simple, let’s assume that the calendar follows the ISO 8601 standard, which includes a year 0 and is proleptic (i.e., applies to dates before the Gregorian calendar was introduced). In Groovy, the appropriate predicate would look something like listing 1.

Focusing just on method names, the procedural style of testing would lead to a single test method, as shown in listing 2a. Anyone unfamiliar with the rule for leap years will be none the wiser. The name tells you nothing about the behavior, only that a single method is being tested. If it fails, all you can deduce is that something about `isLeapYear` (or perhaps `testIsLeapYear`) is incorrect.

It is clear that the behavior needs to be partitioned in some way to clarify what behavior is expected. A first cut at partitioning might be to divide the test cases according to the result of the function—i.e., leap years and non-leap years (uncommonly known as *common years*). The code in listing 2b is an improvement, reflecting a little more of the problem domain in the tests. However, the actual nature of the rule is still unclear. Given that most people believe that leap years occur every four years regardless, this doesn’t clarify matters one way or another and does nothing to correct that belief.

Even when people are aware that leap year determination involves more complexity, they often fail to recall the detail. At

AGILE SOFTWARE DEVELOPMENT TRAINING

Accelerate Your Career & Empower Your Team



BUILD-YOUR-OWN TRAINING WEEK

Maximize the impact of your training by combining courses in one location to create a customized training week. Pair two courses and save up to \$500. For a complete list of courses available, visit www.sqetraining.com or call **888.268.8770** or **904.278.0524** for pairing discount options.



Improve your skills and help your organization increase its performance through targeted high-value training. Delivered by top software consultants, training through SQE Training is one of the best investments you can make to meet your business objectives.

AGILE SOFTWARE DEVELOPMENT TRAINING WEEK LOCATIONS

September 21-25, 2009

Boston, MA

October 12-16, 2009

Seattle, WA

December 7-11, 2009

San Diego, CA

Choose from 4 specialized training courses:

THREE-DAY COURSES (Monday - Wednesday)

- Scrum Master Certification
- Design Patterns Explained

TWO-DAY COURSES (Thursday - Friday)

- Practical Test-Driven Development
- User Stories and Estimation in Agile Development



Let SQE Training come to you. For more information about on-site training courses, contact SQE Training at 904.278.0524 x212 or 233 or 888.268.8770 or email onsitetraining@sqe.com.


```
boolean isLeapYear(int year)
{
    ... // left as an exercise for the reader
}
```

Listing 1

```
testIsLeapYear
```

```
testIsLeapYears
testNonLeapYears
```

Listing 2a

```
testYearsNotDivisibleBy4
testYearsDivisibleBy4ButNotBy100
testYearsDivisibleBy100ButNotBy400
testYearsDivisibleBy400
```

Listing 2b**Listing 2c**

```
testThatYearsNotDivisibleBy4AreNotLeapYears
testThatYearsDivisibleBy4ButNotBy100AreLeapYears
testThatYearsDivisibleBy100ButNotBy400AreNotLeapYears
testThatYearsDivisibleBy400AreLeapYears
```

Listing 2d

```
class LeapYearTest extends GroovyTestCase
{
    void testThatYearsNotDivisibleBy4AreNotLeapYears()
    {
        assert !isLeapYear(1906)
        assert !isLeapYear(2009)
    }
    void testThatYearsDivisibleBy4ButNotBy100AreLeapYears()
    {
        assert isLeapYear(1984)
        assert isLeapYear(2008)
    }
    void testThatYearsDivisibleBy100ButNotBy400AreNotLeapYears()
    {
        assert !isLeapYear(1900)
        assert !isLeapYear(2100)
    }
    void testThatYearsDivisibleBy400AreLeapYears()
    {
        assert isLeapYear(2000)
        assert isLeapYear(2400)
    }
}
```

Listing 3

least part of the reason for this is that humans are good with straightforward rules—and even rules with an exception—but they are not so good at dealing with rules that have exceptions to exceptions (and beyond). For leap years and common years there are four distinct cases, as shown in listing 2c.

This equivalence partitioning delineates the cases explicitly. But there's still something missing: It's like a joke without a punch line. We know the situation we're testing, but not why

or what we're expecting from it. The test cases in listing 2d include the punch line.

The Naming of Names

There is a world of difference between the last four test cases shown in listing 2d and the initial `testIsLeapYear` in listing 2a. These last test cases specify what the “right thing” is and, in their bodies, employ representative examples that demonstrate each case, as in listing 3, and check that the right thing happens. The procedural style merely indicates that some named piece of code is being tested without reference to what is needed or expected. The behavioral style reveals the rules and intentions that inform the code.

Of course, the same test code could have been written within a single test method, `testIsLeapYear`, and grouped into test cases, each grouping headed by a suitable comment. But whenever you are tempted to use a comment, first determine whether the code could be rearranged to express more directly what the comment is trying to say. In this case, don't comment blocks of code, name them as methods. Method-level partitioning also offers more precise and meaningful information when a test case fails.

Ideally test names should be simple statements of fact about what is needed from the code. In this case, we are constrained by the decision to take advantage of JUnit 3, which is conveniently integrated with the Groovy runtime. JUnit 3 requires that all test method names begin with `test`. Unfortunately, this particular verb on its own does little to encourage a specification-driven naming style. If you are writing to this technical constraint, or you still like to think of tests as confirmations rather than as specifications, try using `testThat` as your prefix. This alternative prefix naturally encourages name completion with something a little more statement-like.

Whether or not the code under test is procedural in style turns out to be irrelevant. A procedural partitioning of tests is invariably the wrong choice for GUTs. **{end}**

REFERENCES

- [1] Henney, Kevlin. “Programming with GUTs.” *Better Software* magazine, July/August 2008. www.stickyminds.com/s.asp?F=S13833_ART_2
- [2] Cockburn, Alistair. “The Modern Programming Professional has GUTs.” alistair.cockburn.us/index.php/The_modern_programming_professional_has_GUTs



Which unit test styles have you found to be beneficial and which ones have been problematic?

Follow the link on the [StickyMinds.com](http://www.stickyminds.com) homepage to join the conversation.



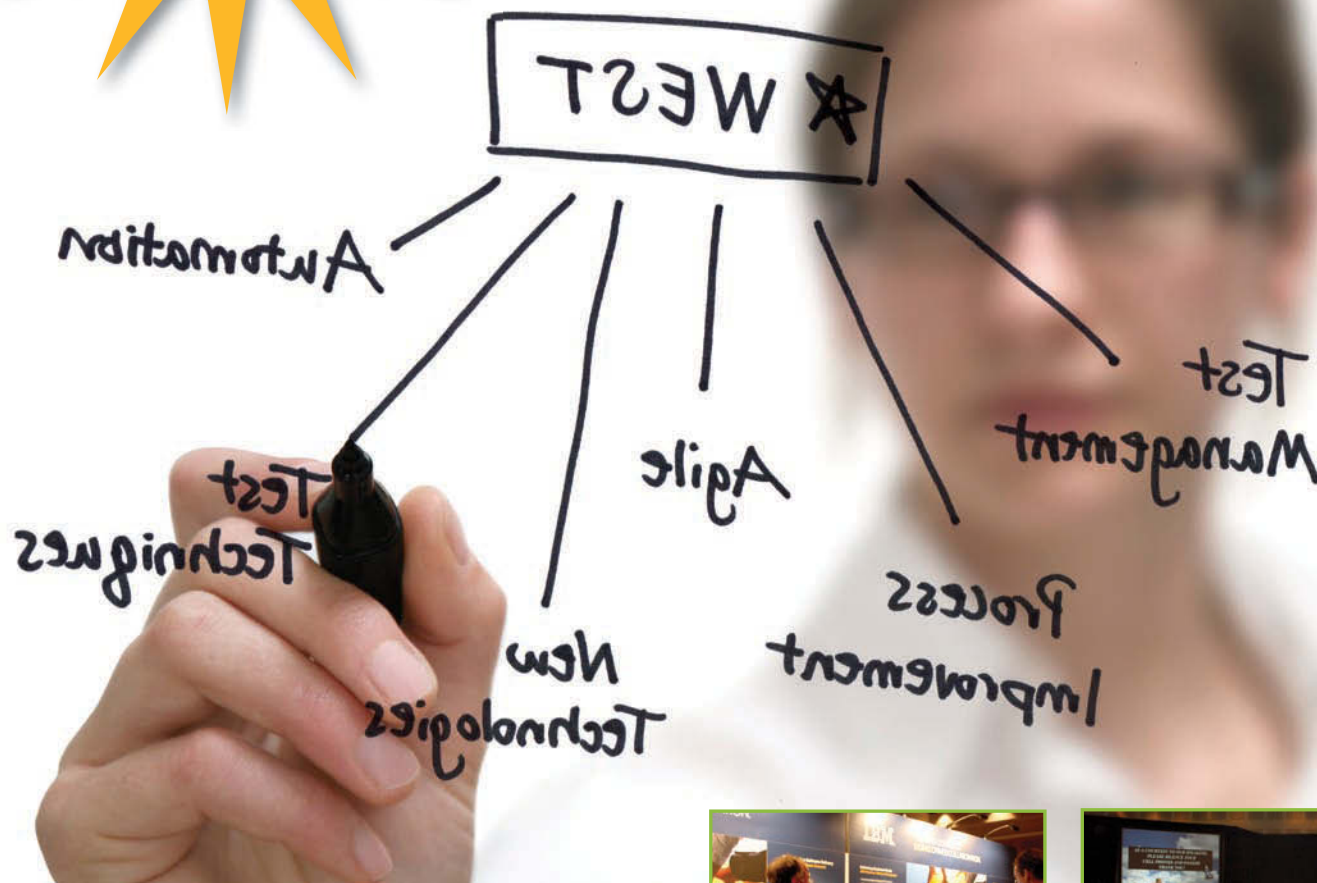
**STAR
WEST**

SOFTWARE TESTING

ANALYSIS & REVIEW

OCTOBER 5-9, 2009
ANAHEIM, CALIFORNIA
DISNEYLAND® HOTEL

The Greatest Software Testing Conference on Earth



You Can't Afford to Miss STARWEST 2009

- 29 In-depth pre-conference tutorials on how to be more efficient and effective
- 37 Concurrent sessions on how to turn challenges into opportunities
- 5 Keynote presentations from top testing experts with experience in down times
- Networking opportunities to see how others are handling economic challenges
- Largest Testing EXPO event to help you find solutions

**99% OF 2008 ATTENDEES RECOMMEND
STARWEST TO OTHERS IN THE INDUSTRY**

www.sqe.com/starwest
REGISTER EARLY AND SAVE UP TO \$300!



New for 2009!
**Testing & Quality
Leadership Summit**



Issues about Metrics about Bugs

by Michael Bolton

In my travels, I've worked with a number of companies that have attempted to assess the quality of their testing—or worse, their testers—using poorly considered metrics. Sometimes the measurement is based on a count of bugs that make their way into the released product (escaped bugs); sometimes the measurement includes another factor, like the number of bugs found before release. To many managers, this kind of measurement has intuitive appeal: If the purpose of testing is to find bugs, then assessing the quality of the testing effort starts with looking at the number of bugs that the testers found or didn't find before the product was released. How could this appealing-sounding metric possibly go wrong?

One answer is found in *reification error* [1]. A bug isn't a concrete thing, like an airplane or an apple. "Bug" is a label for a *construct*, an idea. One idea might be that a feature seems to be missing, another might be that performance is slower than we'd like, and yet another might be that an error message is accurate but unhelpful. A bug could be something missing, or some form of behavior that we don't want to see. To Rapid Testers, a bug is *anything that threatens the value of the product* [2]. Value is multidimensional and subjective. Someone may value a buggy product that she owns over a less buggy product that she considers too expensive. Someone else may reject a product that provides excellent performance, preferring one that is compatible with his other applications. Like "value," "problem," and "acceptability," bugs are not tangible, countable things; they're expressions of part of a relationship between some person and some product [3].

To us testers, often it seems amply clear that some behavior represents a bug. A system crash, an inaccurate calculation, or a mangled record is very likely to be a problem and a threat to the value of the product. A direct violation of a reliable specification is probably a bug, as is a cor-



ISTOCKPHOTO

rupted display or an unexpected, persistent howl from the system's speaker. Yet some evaluations require more subtlety. If the program rejects an input value as "too big" when a reasonable user might disagree, we have reason to suspect a threat to the value of the product—that is, a bug—even if the specification clearly outlines a smaller range of supported values. Is this an intentional limitation or did a business analyst misinterpret or mistype the end-user's requirements? This is why it's so important for testers to change the question "Does this test pass or fail?" to a question that better addresses a possible threat to someone's values: "Is there a problem here?"[4]

One oracle—a heuristic principle or mechanism by which we recognize a problem—might tell us that there is no problem, where another oracle would cause us to perceive a problem immediately. As testers, we may have strong beliefs one way or the other, but it's the project owner who gets to make the decision. That's why our role is not merely to report things that *are* bugs but also to report things that *might be* bugs or that *could be* bugs when viewed through a different set of values.

In addition to reporting bugs, it's also the role of the tester to report on *issues*. Where a bug is something that threatens

the value of the product, for Rapid Testers an issue is *something that threatens the value of our testing*. (Some people call this a concern or obstacle. The concept is important, but the label isn't; call it whatever you like.) If we're uncertain about whether something is a bug or not, that's an issue. If we identify a problem with testability—anything that slows down testing or that makes it difficult to determine whether or not there's a problem—we *may* be seeing a bug, but at the least it's an issue. If we lack sufficient equipment, tools, or training to accomplish the mission in the required time, that's an issue. If a product that we're testing has so many problems that investigating and reporting them dominates the time we have available for finding them, that's an issue, too. We'll come back to that point.

When a product is released or deployed, it's because some person—the product owner—has decided that it's ready to go. That decision should be based on another: Does the product owner have sufficient information to make the ship/no-ship decision? That's a judgment call, based upon not only technical information but also upon business imperatives. A tester is unlikely to have authority over the schedule, the budget, staffing, or market or contractual obligations. So, while the tester helps

to *inform* the decision, he shouldn't be *making* the decision unless he is also the product owner.

For the same reason, the tester should be very careful about asserting that the product has been "adequately" or "completely" tested. A clothing salesman should offer excellent service to the customer as long as she's in the shop, but he can't be held responsible for deciding whether the customer has bought too much, too little, or just enough. If something looks particularly embarrassing or complimentary on the customer, the salesman can assist the customer by pointing it out. If there's some interesting, new piece in the back room, it behooves the salesman to bring it to the customer's attention. But in all cases, the customer—not the salesman—is responsible for deciding what she wants to buy, what services the salesman shall provide, when he has completed his service, and whether the service was adequate.

So it is with a tester and his client. The tester provides information about problems in the product, but those shouldn't be the only items that he brings to the table. The tester may also report on benefits in the product, about comparable products, or about risks. The tester should also be prepared to point out parts of his work that he would recommend covering, but that he hasn't covered. The product owner uses that information, along with all of the other technical and business information, and decides whether she has enough information to issue the order to ship.

This is a good reason to be wary of bug metrics. The project owner is the person who ultimately makes the decision about: what is a bug, whether the known bugs are trivial enough to permit shipment, whether there are sufficiently important open questions such that shipping would be unwise, and whether the business priorities outweigh the technical ones. These decisions will have a profound impact on any attempt to count bugs, either before or after the product is released.

Bugs in the product may inhibit our ability to find bugs. Some problems—blocking bugs—may make it difficult to execute tests by preventing access to parts of the product that require further testing. Other problems—intermittent bugs—may cause test results to be inconsistent, inconclusive, or ambiguous. If there are large

numbers of bugs to fix, programmers may provide us with a large number of builds that we must reinstall and reconfigure—or they may provide us with a single build on which we have to do dozens of fix verifications before we can pick up again with other tests. All of these interruptions—setup, configuration, bug investigation, and reporting—take time away from the design and execution of new tests, wherein we obtain more coverage of the product. So, a buggy product gives bugs more time and more places in which to hide, requires more time to test to the same level of coverage, or both. That's an issue—a very common and very serious issue.

If there are many bugs in the product after release, it may well be that the testers have done less than excellent work. Yet there are many other plausible explanations: a very complex product, inadequate programmer testing, an overly aggressive schedule, a rational business determination by product management that the known problems in the product aren't worth fixing, or any or all of the above. Next time, we'll talk about the risk of metrics based on possible motivations for them: inquiry or control? **{end}**

REFERENCES

- [1] Levy, David. *Tools of Critical Thinking: Metathoughts for Psychology*. Waveland Press, 2003.
- [2] Bach, James, and Michael Bolton. "Rapid Software Testing." www.satisfice.com/rst.pdf
- [3] Bolton, Michael. "It's All Relative" in Fiona Charles, et al. *The Gift of Time*. Dorset House, 2008.
- [4] Bach, James, and Mike Kelly. "Is There a Problem Here?" www.michaeldkelly.com/pdfs/IsThereAProblemHere.zip

Who decides what the criteria for a bug are in your organization? What do you do to make sure that your metrics prompt questions and investigations, rather than drive decisions?

Follow the link on the StickyMinds.com homepage to join the conversation.

mingleTM

AGILE PROJECT MANAGEMENT

Your global project team on the same page
Agile Project Management
Software from the Pioneers of Agile

BANGALORE

Bugfix #31
"In Testing"

BOSTON

Card #54
"Under Development"

BEIJING

Feature #76
"done, Done, DONE"

Provide a shared workspace for Developers, QAs, BAs, Customers & PMs

Capture & Visualize all project activity

Manage XP, Scrum, Lean & Agile Hybrid projects

Get real-time intelligence with burn-down charts, velocity graphs, etc.

Leverage the industry's best User Interface

DOWNLOAD FREE TRIAL AT
www.thoughtworks.com/mingle

STUDIOS
ThoughtWorks

Crash Course in Proficient Presenting

by Naomi Karten

Ben was babbling. He didn't mean to babble. He didn't want to babble. But when he tried to run through his upcoming talk for Natalie, his presentation coach, out came gibberish.

"Sit down," Natalie said gently, "and tell me what you're trying to say." Without a moment's pause, Ben did just that, clearly explaining that his recently completed software project had yielded some valuable lessons that he wanted to share with the group. As he started to outline the lessons, Natalie interrupted him.

"What happened when you tried to say the same thing while you were standing?"

Ben started to look panicky. "I guess I was imagining the packed auditorium I'll be speaking in and I got so nervous, the words fell all over each other."

Natalie laughed. "That's a great image."

"But what am I going to do? I didn't ask to speak at the all-hands meeting. My esteemed director, Mr. You-Have-No-Choice, put me on the agenda. Twenty minutes. Egad!"

"I know it's hard to believe right now, but some day, you'll appreciate what a career-enhancing opportunity he's offered you."

"An opportunity to babble. Lucky me."

"My job is to help you become a confident, babble-free presenter. So let's start by discussing your nervousness."

"If I don't conquer it, I'm sunk," Ben said, sounding like he was sinking fast.

"First of all," Natalie said, "it's a good idea to memorize your opening comments, since that's when you're likely to be most anxious."

"Actually, I thought I'd memorize the whole thing."

"Absolutely not!" Natalie left no doubt that she was opposed to the idea. "It's difficult to memorize an entire presentation, even one that's only twenty minutes long."

Only twenty minutes? Ben thought. *Only???*

"Even if you're able to memorize it," Natalie continued, "all it takes is an unexpected question to make you lose your place. It's much better to think through each point you want to make—you'll have your slides to guide you—and be prepared to address each in your own words. Just memorize the opening few sentences."

"I have a better idea. How about if I just read my presentation?"

"Make that my second 'absolutely not.' Your goal is to sound conversational, not mechanical and robotic. Besides, reading a speech can be risky. Last month, I attended a conference at which a company VP read his presentation. He actually read one line three times before he realized he was repeating himself."

"But if I had a teleprompter ..."

"Dream on! Now, when it's your turn to present, take your place on the platform, take a deep breath, look to the back row of the auditorium, and speak to the people back there. Beginning this way will help you sound confident—and if you sound confident, you'll feel confident."

"I can't imagine feeling confident. Maybe I'll just tell the audience that this is my very first presentation and I'm nervous."

"Not a great idea. Tell them you're nervous and instead of listening to you, they'll be watching for signs of nervousness. And they'll see signs even if they're not there."

"But what if I don't tell them I'm nervous, but I am anyway?"

Natalie looked Ben in the eye. "The key to avoiding an overwhelming level of nervousness is to practice, practice, and practice some more. It's the single most important thing you can do. The better prepared you are, the more confident you'll be."

"So experienced speakers don't get nervous?"

"Actually, sometimes they do. But many experienced presenters find that a mild undercurrent of nervousness actually energizes them and adds enthusiasm to



ISTOCKPHOTO

their presentations. So, to a certain extent, nervousness is your friend."

"Ah, my BFF, nervousness."

"If you feel some anxiety once you're into your presentation, slow down and speak a little louder. Speaking a bit louder masks the nervousness and projects confidence."

"For sure, I'll be the loudest presenter they've ever heard."

"More likely, you'll know your material so well by the time of the meeting that you'll wow the audience."

"So, what else are you going to help me with?"

"I'll review the content of your presentation for clarity and organization. And I'll make sure your slides aren't like the eye-straining ones I see so often."

"Great," Ben said. "I'll have terrific content and perfect slides. But what

STORY LINES

- **Practice is the most important key to becoming a proficient presenter.**
- **A small amount of nervousness can be an energizer. Use it to your advantage.**
- **Feedback from coaches or experienced presenters will help you improve your delivery.**

else can you tell me about actually presenting?”

“A lot of things. For example, people can’t listen to you *and* read your slides at the same time, so when you show a new slide, pause briefly to give them a chance to take it in.”

Ben grinned. “Pausing. That I like. A chance not to talk.”

“Also, you’ll want to make eye contact with the audience.”

Ben looked shocked. “Ohnoooo, do I have to?”

“Indeed, you do. Let me put it this way. I once attended a presentation in which the speaker kept staring at the floor. It was as if he thought that if he couldn’t see us, we wouldn’t be able to see him. Do *not* look at the floor, OK?” Natalie grinned.

She continued. “Another thing. I want you to look like you’re having a good time when you’re presenting. If you look like you’re enjoying yourself, you will convey confidence and your listeners will enjoy listening to you.”

Ben looked dubious. “You make it sound so easy.”

“Not at all. Delivering a polished presentation takes work. But you have some important information to share and your audience will appreciate it. Keep in mind that most people are so fearful of giving a presentation that they’ll be thrilled it’s you up there and not them. They’ll admire you for doing what they’re terrified of doing.”

“So I can’t call in an emergency root canal?”

“Ben, I remember back when I was as anxious about presenting as you are now. I can now speak to audiences of any size and have fun doing it. I know, positively, that you can learn to do the same. And I’ll be delighted to say I told you so. Now, stand up and give it another try.” {end}



Have you heard a presenter who can serve as a role model for you? What aspects of his or her presentation did you find most compelling?

Follow the link on the StickyMinds.com homepage to join the conversation.

Looking for an affordable, yet powerful solution to your automation challenges?

PHANTOM is the SOLUTION!

With today’s economic challenges, you need to provide the best possible products on an ever tightening budget. Phantom provides powerful, affordable test automation to efficiently and reliably catch product defects before they reach your customers.

Visit www.phantomtest.com for a free trial download.



Powerful, reliable, flexible, and affordable. Phantom.

©2009 Phantom Automated Solutions, Inc.



More than 110,000 Certified Testers Worldwide. Why Not You?



There are now over 110,000 certified testers worldwide and it's your turn to become one of them! Get trained and certified in the format that best suits your needs—online, on-site at your organization, or at any of our public locations.

Prepare to take the ISTQB™ Certified Tester—Foundation Level exam and learn the following:

- **Fundamentals of software testing:** Concepts and context, risk analysis, goals, process, and psychology
- **Lifecycle testing:** How testing relates to development, including models, verification and validation, and types of tests
- **Static testing:** Reviews, inspections, and static tools
- **Test design techniques:** Black-box test methods, white-box techniques, error guessing, and exploratory testing
- **Test management:** Team organization, key roles and responsibilities, test strategy and planning, configuration management, defect classification and management
- **Testing tools:** Tool selection, benefits, risks, and classifications

TESTER CERTIFICATION Foundation Level Training



Three Great Ways to Get Certified!

Public Classes

Delivered by top experts in the testing industry, we offer Software Tester Certification—Foundation Level training in over 60 cities! Learn the essentials needed to become a software test professional and understand how testing fits into software development. Learn the fundamental steps in the testing process: planning, analysis, design, implementation, evaluation, and reporting.



On-site Learning

Bring this course on-site and save training and travel dollars! On-Site training is cost-effective and convenient. It focuses on your team's challenges in a small group environment. If you have six or more people to train, consider the advantages of scheduling courses at your location. Find out how to get a complimentary quote. Call Julie or Lily at 888.268.8770 ext. 212 or 233 or email onsitetraining@sqa.com.

eLearning

This new self-paced and highly-interactive online course offers the same expert instruction and valuable content found in SQA Training's instructor-led classes. The course registration includes:

- Powerful multi-media format
- Interactive exercises
- Lesson questions
- End-of-course sample exam
- Complete course manual
- ISTQB™ exam study guide
- 24/7 access for 90 days
- Access to expert test consultants and administrator



For all the details, visit:

www.sqetraining.com/certification



★ SCR

10 TACKLE SOFTWARE DEVELOPERS
USING HIGH-IMPACT SOFTWARE DEVELOPMENT



by Laszlo Szalvay

Today, companies find themselves in the midst of a business world that is continually shaped and reshaped by emerging technology. From Web 2.0 to cloud computing, these new technologies are leading companies to completely reevaluate how they operate and interact with customers. And no one better understands the opportunities and risks associated with a rapidly evolving technology landscape than companies from the tech sector—particularly the software development industry. In short, the ability to respond quickly and flexibly to new business conditions is integral to survival. With that in mind, software developers require a new approach to project management that is nimble enough to keep pace in such a chaotic business climate, but determining which management framework will yield the best results isn't always easy.

One of the new methods currently being employed with success is Scrum, an agile method of project management that uses iterative, incremental work cycles known as sprints to provide frequent opportunities to assess and revise direction. While Scrum's iterative nature ensures that teams are always mindful of the big picture, the framework also facilitates ongoing communication and collaboration to yield high-impact teamwork. When people are connected through frequent check-ins, it enables teams to improve their processes in the long term by becoming the sort of groups that can achieve the near impossible. When an organization develops several of these hyper-performing teams, it creates a path toward sustainably lean operations. In other words, teams like this can do more with less.

An Overview of the Scrum Framework

Before explaining how the Scrum framework achieves all this, it's helpful to begin with what Scrum is and how it differs from other approaches to software development. To begin, Scrum can be defined as a simple management framework for incremental product development. Development is performed by one or more cross-functional, self-organizing teams of about seven people each. These teams use fixed-length iterations called sprints, which are typically fourteen to thirty days in length. Unlike waterfall, the historically dominant method of software development, Scrum

does not assume that all of a project's requirements can be known at the outset or that development can occur in a linear, "single-pass" fashion. Instead, Scrum assumes that more information about the desired product will be collected during development and, to accommodate that, proceeds in sprints, allowing the team to revisit areas of development throughout the lifecycle.

The Scrum method is deliberately designed as a framework—i.e., a lightweight management wrapper that can be applied to existing processes. However, every part of Scrum's minimal framework is essential for realizing its core tenets of facilitating productivity through communication, collaboration, and self-organization. Given its spare structure, it's critical that all of Scrum's roles and processes are observed. Here's a quick overview of Scrum's primary roles and meetings.

SCRUM ROLES

- The *product owner* constantly reprioritizes the product backlog to reflect those items with the highest business value. This individual is responsible for communicating product vision to team members and negotiating sprint goals with them each sprint. He also is responsible for yielding a return on investment and therefore possesses the authority to accept or reject each product increment.
- The *Scrum team* is a cross-functional and self-organizing team

of about seven members that is responsible for delivering a functional product increment each sprint.

- The *ScrumMaster* facilitates team productivity and self-organization by removing impediments that obstruct progress, enforcing Scrum's rules, and ensuring that all Scrum artifacts are highly visible.

SCRUM MEETINGS

- During the *sprint planning meeting*, the product owner and the team negotiate the work that team members will attempt to complete in the next sprint. The product owner is responsible for identifying which work is of the highest priority. Likewise, the team is responsible for committing to the amount of work it can accomplish.
- The *daily scrum meeting* allows team members to deliver updates to one another on a daily basis. Every day, at the same time and place, team members spend fifteen minutes reporting to one another. Each team member reports to the rest of the team what he did the previous day, what he will do today, and what impediments block his progress.
- The *sprint review meeting* occurs at the end of the sprint. At this meeting, the team demonstrates the functional product increment it has developed and the product

owner either accepts or rejects the work, based on the previously negotiated agreement. This is an opportunity to “inspect and adapt”—that is, to examine the product’s progress and revise direction, if necessary, for future sprints.

- At the *sprint retrospective*, the team inspects and adapts its own processes. During this meeting, the team reflects upon its performance in the past sprint and brainstorms ways to improve going forward.

A Mix of Structure and Flexibility Empowers Teams

Scrum’s iterative and incremental approach to development benefits teams. During each sprint, a team commits to a defined amount of work, which it must complete over the course of the sprint. These action items—called *product backlog items* in Scrum—are negotiated between the team and its product owner. However, once work is assigned for the sprint, the product owner cannot give the team additional work until the sprint concludes and planning takes place for the next one. The team is empowered with the autonomy to determine how it completes the work and in what priority it will tackle it. In Scrum, this is called *self-organization*.

For traditional project managers who try Scrum, resisting the temptation to micromanage can be a considerable challenge. Learning to trust that a team will self-organize effectively to complete its sprint goals may be even harder. But this aspect of the Scrum framework carves out a stable, uninterrupted stretch of time for teams to complete work. Given the inherent volatility of writing software, Scrum endeavors to minimize the thrash of the variables it can control—in this case, the disruption of an overly demanding product owner. Moreover, the iterative nature of sprints allows teams to establish a rhythm, creating a kind of routine velocity for each sprint.

As team members become increasingly comfortable with one another over time, they learn how to work together more efficiently. Such hyper-productive teams are the key to organizations’ maximizing their potential.

In another departure from the waterfall method of project management, Scrum possesses the unique characteristic of allowing teams to begin work on a product without knowing all of its requirements. Where waterfall historically asserted that all requirements could be known absolutely at the beginning of a project, Scrum believes that, just like work, requirements can be added incrementally as development progresses. With waterfall, teams waste time and energy attempting to identify every feature of the product before building begins. Not surprisingly, they typically fall into a state of “analysis paralysis,” in

the product scope through each meeting with the customer.

How Scrum Generates Business Value

One of the greatest advantages of Scrum over traditional project management models is that Scrum projects are developed iteratively and incrementally. Because teams work in repeatable, timeboxed work cycles, they add functionality to products with each successive sprint. To ensure that the team progresses toward completion, Scrum mandates that teams produce a “potentially shippable increment” every sprint. This means teams always have a demonstrable product to show a client. It might contain only barebones functionality at first, but it forces teams to make tough decisions and hit the ground running. Equally important, it produces a work in

progress to share with the customer, who can provide direct feedback. In fact, there’s a strong business case for building frequent customer interaction into the Scrum process. Customers typically

“Given the inherent volatility of writing software, Scrum endeavors to minimize the thrash of the variables it can control...”

need to interact with a piece of software to better articulate their vision for it. This phenomenon is called “I’ll know it when I see it,” or “IKIWISI” for short.

Of course, the true test of any project management strategy is its ability to generate business value. Scrum accomplishes this in a number of ways, all of which contribute to the end goal of creating a product that meets the customer’s expectations. As stated above, Scrum brings the customer and the development team together as often as every sprint. Because the customer is involved throughout the development process and is consulted once per iteration, the team can never deviate very far from the customer’s vision. Even if, for example, a development team takes the product in the opposite direction, it can’t get too far off track because the sprint is timeboxed. Compared to waterfall development, in which development proceeds in a linear, unchecked fashion, Scrum’s abbreviated feedback loop guarantees that the team always circles back to the customer to

which the requirements-gathering phase becomes so daunting that the team simply does nothing at all. Scrum, on the other hand, knows that defining certain features may only be possible once other, more basic, functionality has been built.

Of course, Scrum still asks teams to produce a functional product increment each sprint, which means that, even during that initial sprint, the team must settle on some foundational functionality to build immediately. That first sprint forces teams to make hard decisions and begin work, thereby saving the time and money that would have been wasted futilely trying to wrap their arms around as-yet-unknown requirements. In software development, this is a radical step toward responsibly conserving time and money. By acknowledging the limitations of what it can know at the outset, the development team is able to focus on those requirements it is certain the client expects. Thus, the team immediately begins building essential functionality, adding other features to

“...organizations must leverage Scrum’s emphasis on

transparency and communication

to manage employee anxiety about the transformation.”

check the team’s progress against customer expectations during the sprint review.

Because requirements are added incrementally in Scrum, it is possible to delay certain decisions until the last responsible moment. This provides customers with the flexibility to adapt—even to late-breaking circumstances—without breaking the bank. Imagine a competitor rushes to market a rival product that includes new functionality absent in your product. Even though your product is nearly complete, its scope can quickly be revised to incorporate this feature. With the product’s base functionality already in place, another feature simply represents another increment of work. By contrast, waterfall’s exhaustive, up-front requirements would have locked the team into developing a product that could not compete in the marketplace. After all, this is why Scrum believes requirements should be gathered throughout the development process. This ability to make last-minute revisions to scope at a low cost is what makes Scrum so “agile.” Even when it appears that a competitor’s product has already won, Scrum allows developers to respond quickly, nimbly, and inexpensively by extending the product lifecycle by a few iterations. Instead of a product that is rendered irrelevant by the market, the team simply has an underdeveloped product, with the additional functionality, which can still go to market and compete.

Challenges

While Scrum addresses many of the shortcomings present in other development methods, it does present a few challenges of its own. However, the majority of these impediments are limited to

the transformation process, in which an organization first transitions to Scrum. Because Scrum asks teams to adopt drastically different working habits, it is often met with a significant culture clash. It is not uncommon for individuals to fear change, especially in the workplace, and the effects of change during a Scrum transformation can range from mild concern about learning to work differently to an outright refusal to comply. Therefore, organizations must leverage Scrum’s emphasis on transparency and communication to manage employee anxiety about the transformation.

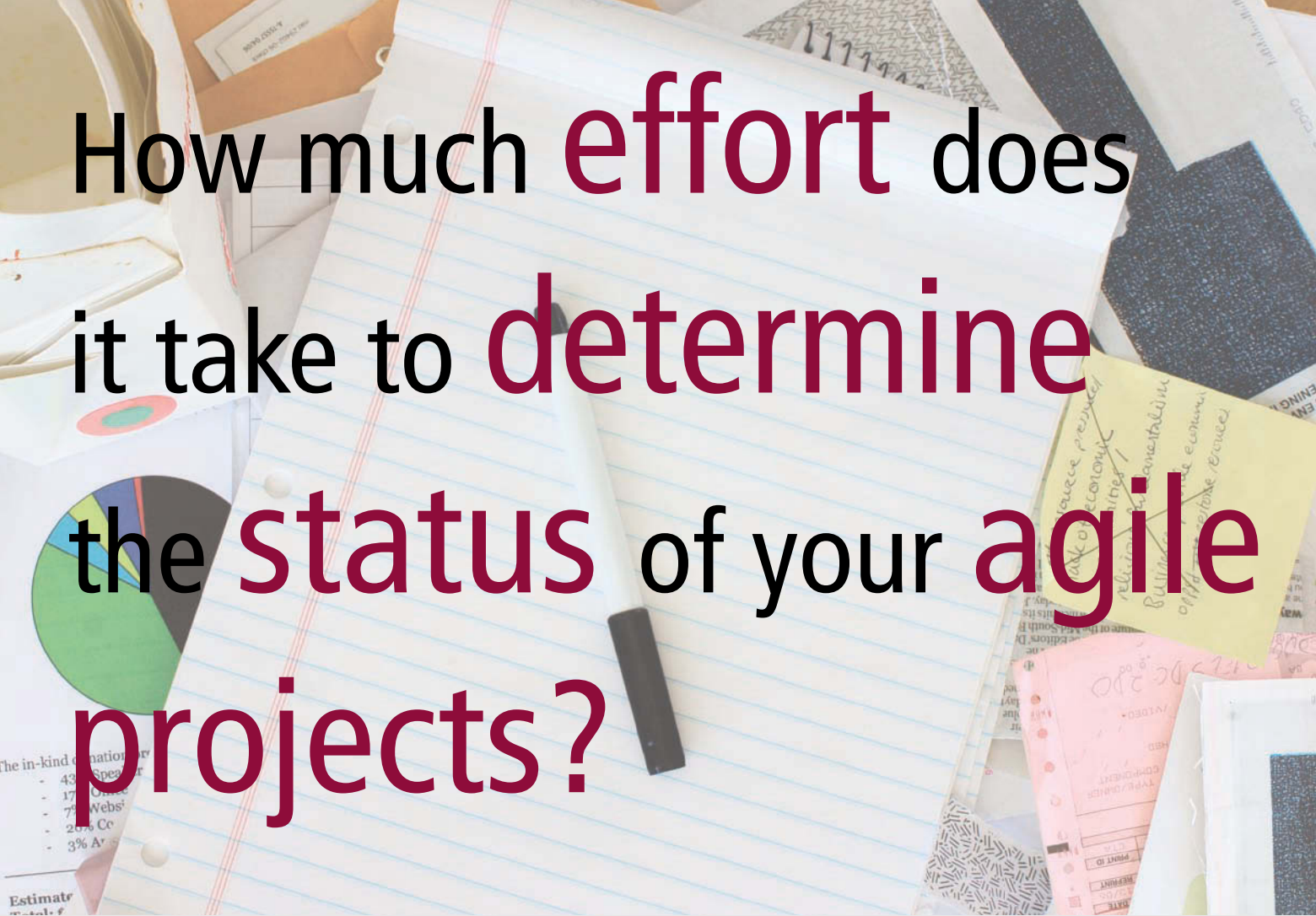
In addition to some cultural resistance, organizations commonly observe some administrative roadblocks, as well. Given Scrum’s emphasis on teamwork and the relatively small number of roles it contains, human resources departments must revise existing career paths and incentive programs to ensure that the organization’s values align with Scrum’s. The good news for organizations facing the challenge is that Scrum’s incremental approach to development can be applied to situations like these, as well. Often, a “transformation team” is utilized to make sure that Scrum’s principles and processes are implemented at both the team and organization levels.

Conclusion

In the end, Scrum succeeds because its emphasis on communication brings people together to share problems, brainstorm solutions, and realize a vision. Fueled by this kind of engaged collaboration, teams form bonds through shared commitment and past successes, which, in turn, push teams to levels of productivity they never dreamed they could achieve. To facilitate this communica-

tion, Scrum believes that the customer, who understands what the product should become, and the development team, which performs the work, should meet frequently to make sure that they build the *right* product. Because a product’s lifecycle is divided into increments, an opportunity to pause and assess the product’s direction against real-world conditions is never more than a single sprint away. And though traditional project managers may be frightened by Scrum’s belief that development can begin before all requirements are known, they shouldn’t be. In fact, this approach to development is actually rooted in reality, taking inventory of empirical data to help the team make informed decisions as information becomes available. Waterfall effectively ignores the fact that, by the time a product is ready to ship, the thrash of the real world has rendered a team’s work irrelevant. Scrum actively guards against this wasted effort by encouraging teams to dive in and build requirements as they become known.

When a team’s activities are determined by customer feedback and empirical data, there’s a greater likelihood that the team will be able to proceed with confidence and clarity. And when teams can progress with such measured intentions, they often outperform their own expectations. Especially for tech companies, who must remain attuned to how new technology impacts their business, Scrum’s combination of frequent communication and attention to real-world factors enables teams to build products that customers actually want, while operating in a sustainably lean fashion. And that translates to success—no matter what business you’re in. {end}



How much **effort** does it take to **determine** the **status** of your **agile** projects?

With the VersionOne agile project planning and management system, the answer is: almost none. Our centralized platform consolidates up-to-date project data to give you comprehensive project visibility at the click of a button. Not only will your team save time with our built-in agile process workflow, they'll also experience enhanced collaboration and higher productivity.



VersionOne.
More visibility.
Less overhead.

Let us help you simplify and accelerate your agile software development.

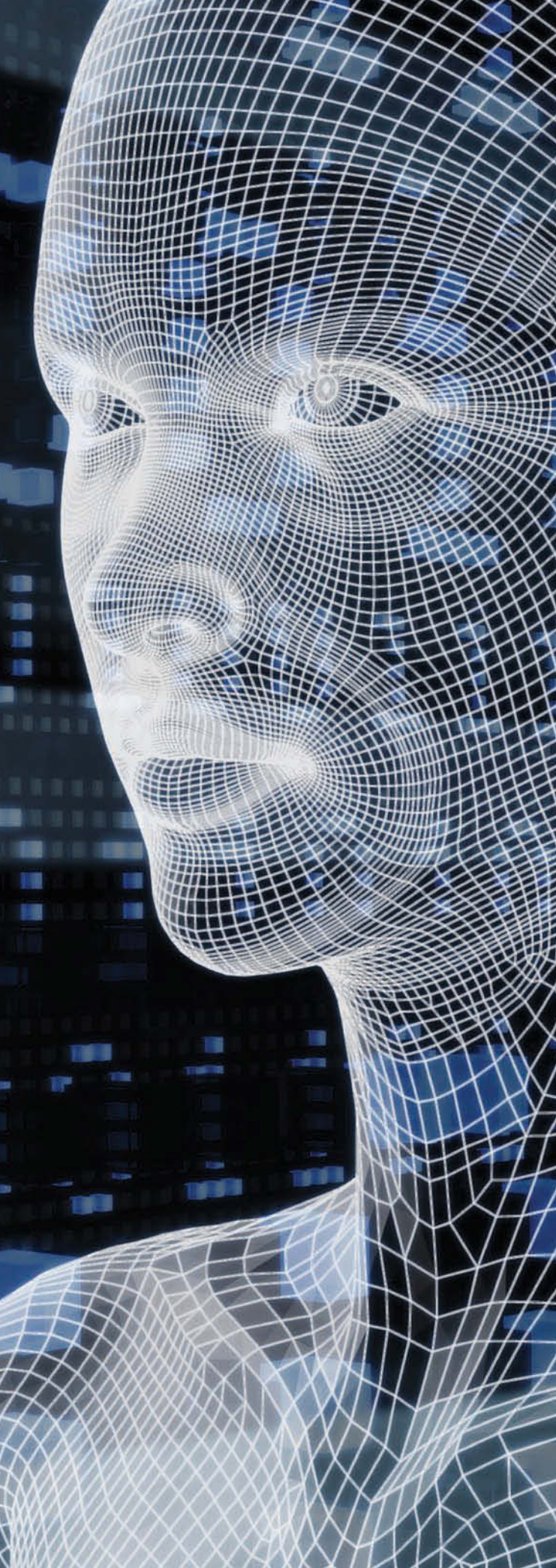
For a free 30-day trial, go to
<http://www.versionone.com/bettersoftware>.

The background is a dark blue field filled with numerous translucent blue cubes of varying sizes, some appearing to float or move. In the bottom right corner, a portion of a white wireframe sphere is visible.

VIRTUAL REALITIES

**BEST PRACTICES
AND COMMON PITFALLS
OF ADOPTING VIRTUAL LAB
AUTOMATION**

BY IAN KNOX



VIRTUALIZATION is a ground breaking technology that promises quantifiable benefits for application development and test organizations, including faster lab deployment, less manual setup work, greater resource flexibility and utilization, and easier reproduction of defects.

However, adopting virtualization in a development or test organization is not without issues. Often, it's not obvious whether to build a custom virtualization framework or make a strategic bet and implement a full virtual lab management solution, complete with automation and a pool of centralized hardware.

This article discusses the software quality challenges that application development teams commonly face and describes the best practices and common pitfalls associated with virtual lab automation (VLA) adoption. Finally, this article outlines the steps to evaluate a virtualization solution for your test organization. The StickyNotes provide links to resources to help you get started

Virtualization Challenges

Many test teams use virtualization by building a proprietary framework using scripting. This can provide many benefits, including faster machine deployment times, better utilization of hardware, and the ability to snapshot and restore images.

Many organizations have used this approach successfully, especially with ad hoc or simple test frameworks. However, as many teams are learning, it can soon become a significant effort to develop and maintain scripts and a library of machine images. Additionally, without implementing a virtual LAN, it's not easy to deploy and manage multi-machine configurations in an isolated network. Usually, there is no user interface to manage the test lab, which limits the control users have over the lab environment. The cost of administration can quickly warrant implementing a more robust solution, and many organizations are investigating VLA to lower the overhead costs of a custom solution.

A manager in charge of application test needs to determine the best way to apply limited resources to achieve the goal of delivering a reliable, performing application. Given the nature of modern distributed environments, this job is becoming more and more difficult due to:

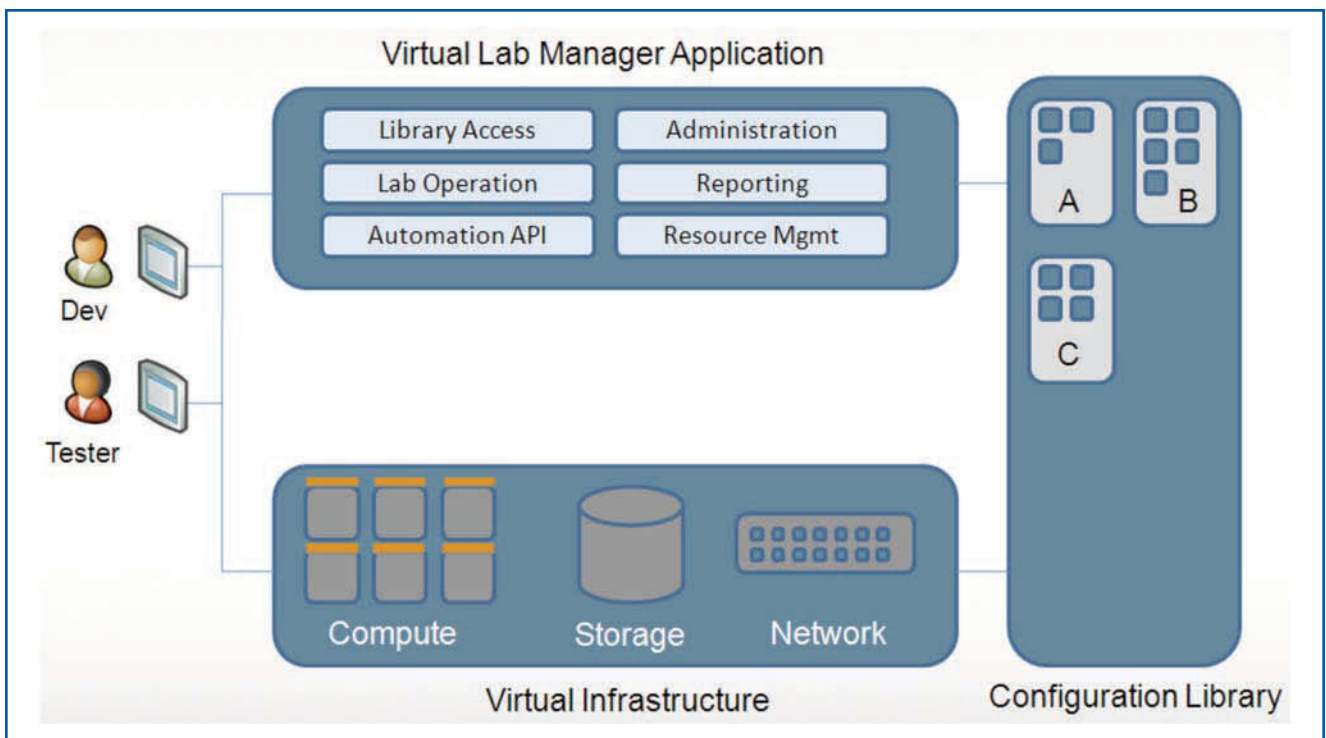


Figure 1: A typical VLA solution

- **Increasing complexity**—Building a lab environment to support testing is a painstaking task for a typical distributed architecture. Implementing test environments that mirror production as closely as possible requires that machine, network, and application settings be carefully configured to ensure that environmental issues are found before deploying to production.
- **Resource constraints**—Budgets are shrinking, and procuring physical hardware, storage, and network resources for test environments is costly and often difficult to justify given low utilization. Since applications often require testing on multiple operating system versions, language variants, browsers, and devices, achieving the optimum balance between adequate test coverage and acceptable risk is difficult.
- **Productivity bottlenecks**—The setup and tear down of labs is usually a time-consuming, manual process. This lab provisioning overhead is costly and reduces the time teams can spend testing an application.
- **Faster cycle times**—The broad adoption of agile development practices has put pressure on QA teams to reduce test cycles and work it-

eratively to deliver software.

- **Communication issues**—Developers often spend an inordinate amount of time trying to reproduce and debug defects reported by the test team. If an application state is difficult to reproduce, it can mean hours wasted diagnosing an issue.
- **Globally distributed teams**—The growing trend of using offshore testing resources compounds the problem of sharing consistent environments across teams and facilitating effective team collaboration.

Given the rise of virtualization, application development managers and test professionals are rethinking tooling, practices, and skills to help solve these ongoing challenges. Many have already experimented with virtualization in their lab environments and are now realizing a virtual lab automation solution is necessary to overcome new challenges that a virtualized environment brings.

Best Practices and Common Pitfalls

BEST PRACTICE #1: UNDERSTANDING VLA CAPABILITIES AND LIMITATIONS

Virtual lab automation is the industry term that has been coined to describe

a new breed of tools and test practices that uses virtualization technology to automate development and testing labs. A VLA solution, like that shown in figure 1, can include some or all of the following capabilities:

RESOURCE POOLING AND PROVISIONING

Resource pooling enables different teams and individuals to share processing power, storage, and networking infrastructures, thereby increasing utilization and availability of resources and reducing costs. In conjunction with resource pooling, an orchestration and provisioning process allocates and releases resources as needed.

MULTI-MACHINE CONFIGURATIONS

Virtual machine images are the containers that enable isolation of operating systems and applications from physical resources. A configuration is a group of virtual images that defines a complete system, including network and storage characteristics. For instance, a configuration could consist of multiple Windows Vista client machines, an Oracle database server, and a WebSphere application server. Configurations can easily be created by combining virtual machines through a user interface. A configuration is a very useful concept for testing

Test Types	Virtual Lab Support
Unit Testing	●
Functional Testing	●
System Testing	●
Integration Testing	●
Unit Testing	●
Performance Testing	○
Regression Testing	●
Localization Testing	●
End-User Acceptance Testing	●
Appliance / Hardware-Specific Testing	
Key: ○ Partial capability ● Full capability	

Figure 2: Test types supported

teams because it allows a whole system to be defined and isolated in a test environment. Virtual networking enables copies of the same environment to be run in parallel and allows the emulation of production environments during the test process.

CONFIGURATION LIBRARY

The configuration library allows a team to manage and organize virtual images and configurations. Standard builds and images can be created and made available to development and testing teams to save hours of set-up time and environment configuration. Additionally, the library is used to store new configurations that are cloned or created as part of a test.

SUSPEND, SNAPSHOT, AND RESTORE

The ability to suspend the complete state of a multi-machine configuration and make a snapshot—a copy at a point in time—is a major benefit of virtualization. This is especially useful for application development teams because, when a defect is found, a configuration snapshot can be taken at the point of failure and a link to the configuration added to the defect report. Instead of spending hours to reproduce the system state to isolate the defect, a developer can restore the configuration and start debugging within minutes.

SCHEDULING AND RESERVATIONS

Many in-house virtual lab implemen-

tations have a fixed pool of resources for teams to share. The scheduling and reservation functionality allows development and test environments to be reserved ahead of time.

REPORTING AND MONITORING

Reporting modules allow users and administrators to manage usage and quotas and to determine whether the system resources are being used optimally. Monitoring enables diagnosis of the system health, including CPU utilization, storage performance, and network usage.

AUTOMATION API

Automating a test lab almost always involves integrating tools and test processes. An automation API enables teams to create test environments automatically as part of the build process and initiate automated test runs once a new build has been deployed. An automation API is typically made available through a Web services interface.

ADMINISTRATION AND SECURITY

Administration and security features often include user and quota management, project creation, permissions, and authentication. Remote access to the system—for instance, for an outsourced vendor—is usually managed through secure connections via encrypted protocols and virtual private networking.

These capabilities are generally useful to the vast majority of development and test teams. However, every team is dif-

ferent, and determining whether some or all of these capabilities are needed is important to consider. Any manager who has implemented automated tools, whether for development or testing, knows that there are some tools for which the effort required to implement can far outweigh the benefit. The same is true for virtual lab automation.

COMMON PITFALL #1: MISUNDERSTANDING TEST TYPES THAT ARE SUITABLE FOR A VIRTUAL LAB

Almost all VLA solutions utilize a standard infrastructure and support a wide range of test scenarios. Hardware and network characteristics can be configured easily through the Web interface to specify number of processors, amount of memory, and network settings of machines in a configuration.

Typical customer test scenarios, shown in figure 2, include unit testing, functional testing, system testing, integration testing, and load testing of applications. However, there are a few uses where a VLA solution is not recommended. These include tests that require specific hardware access (e.g., BIOS driver tests) and some types of performance and stress testing (e.g., a test of application performance on a specific hardware profile).

BEST PRACTICE #2: DETERMINE THE RIGHT IMPLEMENTATION APPROACH FOR YOUR ORGANIZATION

There are two basic approaches for adopting a VLA solution: 1) purchase an installed package, and 2) use a cloud-based virtual lab service. Each approach has its advantages and disadvantages.

A number of vendors offer packages for VLA. Many of these solutions offer most or all of the capabilities described above. They typically have been adopted by large enterprise organizations where the expense, time, and organizational changes required to build a centralized lab are worth the effort. Voke, an IT analyst firm, estimates a VLA solution can deliver a 25 to 50 percent reduction in hardware needs and an average time savings of three days to deploy a lab environment.

But, a few important areas should be considered before deciding to go with an in-house VLA solution. First, implementing an automated test lab requires

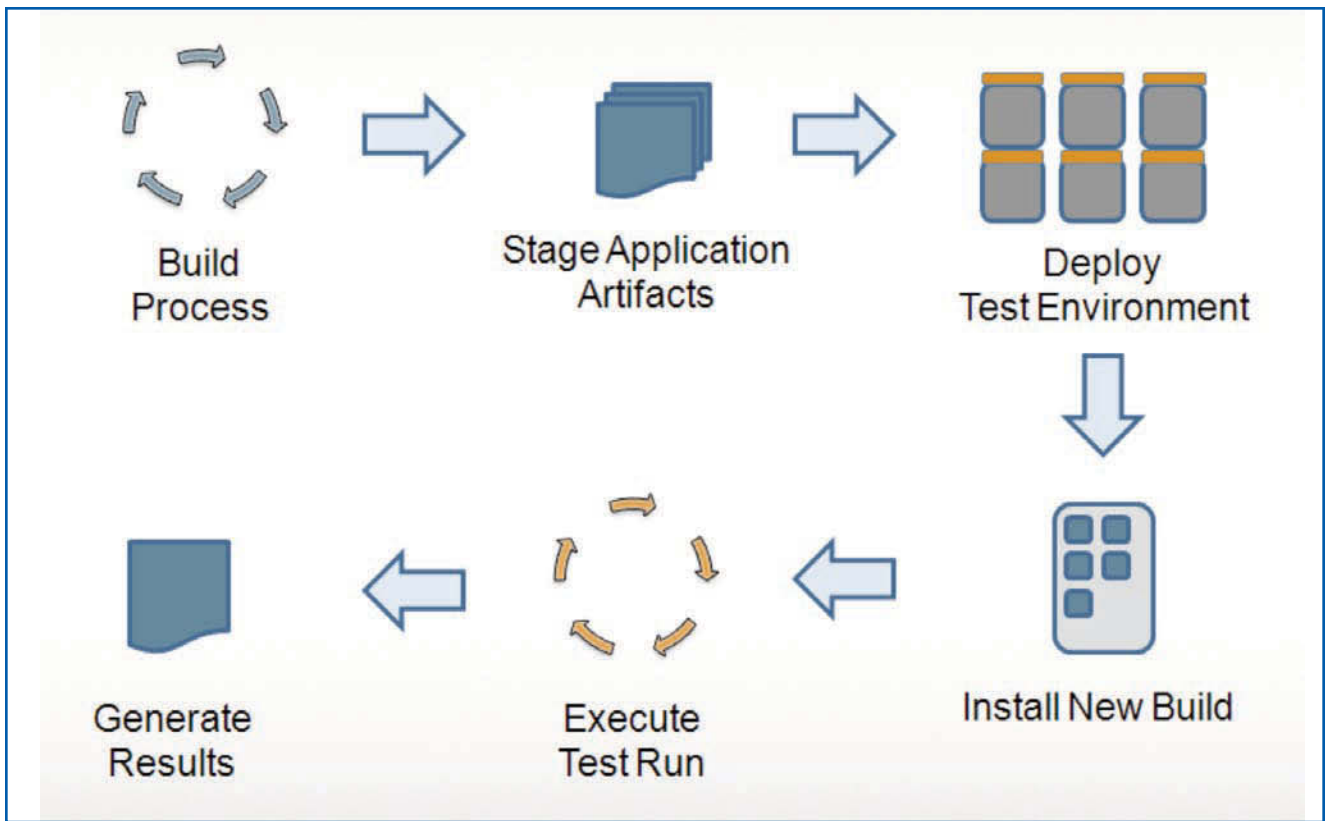


Figure 3: Typical lab automation process

not only a large, up-front investment in lab hardware and software but also significant effort to implement, configure, and train lab personnel. These expenses are especially difficult to justify if budgets are under pressure.

Second, implementing an in-house virtual lab solution requires assigning skilled IT resources for administration and virtual image population and maintenance. Unless a lab reaches a critical mass to cover multiple development and test organizations, this administration overhead can be prohibitive.

Finally, although an automated virtual lab improves resource utilization, there are going to be resource conflicts between teams unless an expensive pool of infrastructure is purchased that covers peak demand. This means some groups will still wait for resources, reducing their effectiveness and increasing delivery risk.

An alternative to an in-house virtual lab approach is a cloud-based solution or “virtual lab as a service,” which solves many of the issues and risks associated with an in-house implementation. For example, there are no up-front investment costs, and you can scale the infrastructure up and down according

to your needs. Furthermore, the service provider handles the administrative costs associated with running the lab. A hosted lab can be integrated easily with onsite assets using a virtual private network (VPN) connection, and a customer typically will pay only for the hourly usage of the lab machines when in use, thus eliminating the expense of test machines sitting idle.

COMMON PITFALL #2: IGNORING INDIRECT COSTS

When determining the total cost of ownership (TCO) for each approach, it's important to remember to include the costs for internal IT support, as well as the more obvious hardware and software capital expenditures. Often, much of the ongoing cost for a lab involves IT administration. Also, indirect costs—such as time savings gained by development and test team members' using the capabilities of a VLA solution—should be included.

BEST PRACTICE #3: AUTOMATE YOUR TEST LAB OPERATIONS

One of the major benefits of a VLA solution is the ability to integrate with a build process and testing framework

to enable an automated workflow, as shown in figure 3. Typically, this workflow is enabled through scripting or tool support in the build server or automated testing tools. Almost all VLA solutions—both in house and hosted—offer an API to enable such integration.

Populating an asset and configuration library enables quick deployment of standard environments. Many environment setup tasks can be automated to avoid manual user intervention. For instance, as part of a nightly build, new virtual machine configurations can be deployed automatically and software builds and patches installed in preparation for a test run the next day. In addition, using the VPN functionality found in a typical VLA solution, a test environment can be deployed to mimic a production environment as part of the automation process.

COMMON PITFALL #3: FALLING FOR THE “SUNK COST” FALLACY

Some IT organizations invest too much time in building their own virtual infrastructure and automation. Often citing work already completed on a virtualized platform, these organizations invest further in areas such as remote ac-

cess for distributed teams, storage management, and networking—areas that come standard with a VLA solution. Instead, these organizations should move to adopt a package or virtual lab service that will save costs in the future.

BEST PRACTICE #4: ENABLE TEAM COLLABORATION WITH USER PERMISSIONS AND PROJECTS

Every VLA solution offers the ability to specify user access levels and permissions. Typically, an administrator has access to the entire lab, team leads can create new projects and environments, and individual testers work as part of a project and have access only to the resources they need.

Once user access control has been specified, administrators and team leads can enable or restrict access to resources through the virtual lab user interface. This is especially useful for projects where outsourced vendors are utilized. A project can be created with the environments to be tested, and outsourced testing professionals can be given access only to the resources required for a given test run. If an organization is using a hosted lab, it's easy to ensure that the test environment is isolated from the corporate network.

With environments stored in a configuration library, replication of defects becomes much easier as both development and test teams share the same environments on the same virtual infrastructure.

COMMON PITFALL #4: NOT RESTRICTING ACCESS TO MASTER CONFIGURATION IMAGES

There will be a set of master, or “gold,” configuration images that are commonly used to create environments. These include standard desktop images, application builds, and server images. It's important to ensure these images can be accessed only by a lab administrator and not accidentally modified by a developer or tester.

BEST PRACTICE #5: OBTAIN TEAM BUY-IN WITH HIGH-IMPACT, LOW-EFFORT CHANGES

Once implemented, training staff members on the use of a VLA solution is the first step to encourage adoption.

However, demonstrating how it will help make their jobs easier is equally important. By choosing a few high-impact areas on which to focus and securing some quick wins to improve productivity, your team is much more likely to adopt the new solution.

One of the most obvious areas on which to focus is populating the configuration library. If you have chosen a hosted VLA solution, this will be prepopulated and you will only need to update base virtual machine images to match corporate images. This will enable IT operations and test leads to quickly assemble and deploy new environments, reducing the time to test and simplifying environment configuration.

ONCE IMPLEMENTED, TRAINING STAFF MEMBERS ON THE USE OF A VLA SOLUTION IS THE FIRST STEP TO ENCOURAGE ADOPTION.

Another high-impact, low-effort change is to ensure snapshots of virtual machine configurations are captured for new defects. This enables testers to communicate defects more easily—some have called it a “screenshot on steroids”—and developers to isolate problems in a fraction of the time it previously took.

COMMON PITFALL #5: INADEQUATE TRAINING OF OUTSOURCED TEST TEAMS

Organizations typically will roll out training programs to their staff but may forget to do this for outsourced test teams, especially when new service providers are added after a VLA implementation. Conducting short training sessions will ensure outsourced teams are equally productive with a VLA solution.

Evaluating a Virtualized Solution

As part of evaluating approaches for adopting virtualization in your development and test organization, I recommend the following steps:

1. **Determine the capabilities your organization needs**—Build a requirements matrix and determine the types of testing and usage patterns typically seen in your organization. Consider operating platforms and hypervisor vendors used in your environment as well

as the current skills of your IT operations and test team.

2. **Explore the potential solutions**—Evaluate an in-house package versus a hosted virtual lab service.
3. **Evaluate TCO**—Build a TCO model. Be sure to include software, hardware, implementation, and administration costs. Also, try to estimate the indirect costs associated with each option.
4. **Conduct a trial project**—Conduct a proof-of-concept or trial project using your short list of solutions.
5. **Implement solution and refine your test process**—Once you've tested and implemented your solution, evaluate your current test practices, update them to reflect the new virtual lab capabilities, and invest in training your team before rolling out the solution.

In summary, I'd recommend exploring VLA if you are experiencing challenges with testing bottlenecks caused by inefficient lab provisioning or utilization of testing hardware, or if you are struggling with sharing test environments across the team or virtual machine sprawl. If you want to experiment with a low-cost, low-risk approach, first explore a hosted (or cloud-based) virtual. From there, you can determine whether an on-site lab is worth the investment for your organization. Regardless of the option you choose, virtualization and VLA are mature technologies, so now's the time to move to the next generation of testing infrastructure! **{end}**

Sticky Notes

For more on the following topic go to www.StickyMinds.com/bettersoftware.

■ Resources

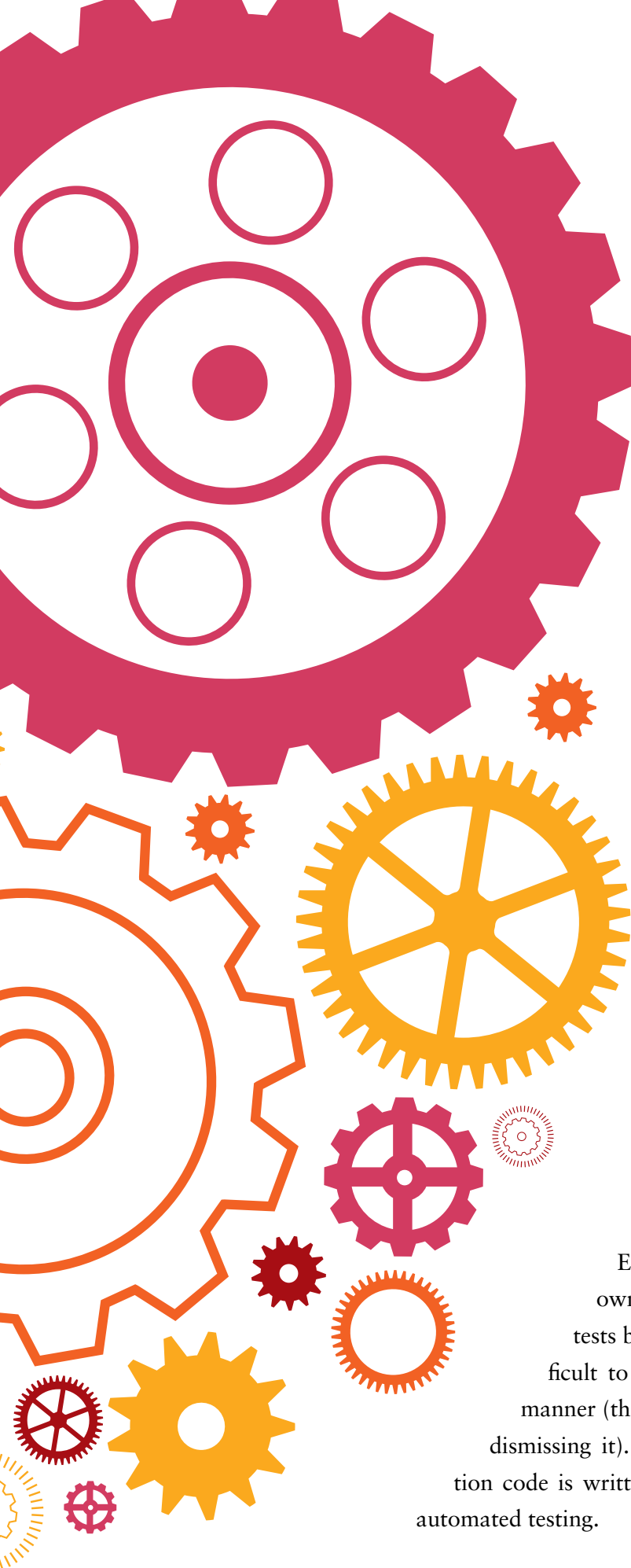


What to Expect When You're Automating Testing

Test-last Tips from an Agile Expert



by Daniel Wellman



Congratulations, you've decided to start writing automated tests for your application. Maybe tests are a new requirement for your team, maybe you've been burned by bugs that keep reappearing, or maybe you were just curious about the buzz surrounding automated testing. However you got to this point, a good suite of automated tests can make your development life more productive and peaceful.

If you are a developer, perhaps you've read an introductory article on testing with JUnit (or NUnit, Test::Unit, or your programming language's flavor of the xUnit test framework), and you understand the syntax and fundamentals of writing tests. But going from test-driving a stack data structure (a typical book example) to testing your living, complex production application can seem like a daunting challenge. In this article, I'll suggest what to start testing in your application, how to get started, and some problems you may encounter along the way.

It's important to note that writing automated tests can be practiced in any software process or methodology, whether it's Scrum, waterfall, RUP, Extreme Programming (XP), or your organization's own custom blend. While XP practitioners write their tests before the production code, this practice can be difficult to start—and not everyone prefers to work in this manner (though I'd encourage everyone to give it a try before dismissing it). Test-last development—testing after the production code is written—is a way that many teams start and practice automated testing.

What Do I Test?

Like starting anything new, it can be difficult to decide how and where to begin adding tests. While sitting down and writing tests for the first code you see may feel productive, there are some strategies to get more immediate value out of your tests. When starting with no existing tests, you want to get the most value out of your time and effort. You want to avoid writing tests for code that never has broken and probably never will break. Here are a few questions I ask to discover some possible starting points:

- Are there any existing or recently fixed bugs?
- What features have you just finished?
- What are you planning to work on next?

ARE THERE ANY EXISTING OR RECENTLY FIXED BUGS?

Tests that expose a bug are the most immediately valuable tests. These demonstrate a real fault in the system, provide feedback for when you have finished fixing the problem (the test passes), and act as an automated alarm if the defect ever gets reintroduced.

If you have just found a bug, write a test that reproduces the failure. Fix the bug, rerun the test to see it pass, and refactor the code to clean it up. This rhythm is the Holy Grail of test-driven development—the “Red-Green-Refactor” cycle.

What do you do if you don’t have any bugs (that you know about)? Pick a bug that you just fixed or one that was a real zinger. Write a test for that bug—but here’s the catch—roll back the production code to a point where the bug still existed. Now run the test to confirm the test exposes the bug, then restore the production code to the current working state. This step is critical—you want to make sure you write a test that actually catches the bug. When you have code that already works, it’s easy to write a test that never actually fails, even if the bug gets reintroduced.

WHAT FEATURES HAVE YOU JUST FINISHED?

When do you best remember the code

you’ve written? Right after you write it! Use that mental clarity to write tests that exercise core aspects of the feature, using the tests to document how things are supposed to work. Now is the perfect opportunity to write some executable documentation that demonstrates any unusual corner cases of the particular business rule you’ve just implemented. Then when you need to modify it in six months, you’ve left some breadcrumbs to remind you of all the particular nuances.

One other practical reason to work on code you just finished is that team members sometimes get grumpy when people change their code or even insinuate that it might be wrong and require testing (perish the thought!). If you’re on a team like this, practice introspection and test your own code; once you do it enough, your teammates might take notice.

Another good reason to test new code is that it may be easier to get the business sponsors to buy-in on the effort. If you tell a business user that you want to spend some time defect-proofing a feature, he’ll probably like the idea. If you tell him you want to work on some other feature that’s not being worked on anymore, he’ll probably balk at the idea. Instead, focus on the current features being developed.

WHAT ARE YOU PLANNING TO WORK ON NEXT?

When you add new features to an application, there is some risk of breaking existing functionality. This is a fact of life, but tests can help. There are a couple of options to consider: add regression tests to prevent introducing new bug, and to understand how some existing code works.

If you’re about to start a new piece of functionality, use your current knowledge of the system to figure out what might break and write tests to cover those cases. Once you complete your task, use these tests to ensure you haven’t broken anything. This is especially true if you’re about to start refactoring code. Make sure you write tests to ensure you don’t accidentally change the code’s behavior.

But what if you didn’t write the code

in the first place and don’t really know what it’s doing? Write tests to demonstrate and discover the code’s behavior. In his book *Working Effectively with Legacy Code*, Michael Feathers calls these *characterization tests*. The theory is that if a system is working, the correct behavior doesn’t come from some requirements specification document; it comes from whatever the code is doing right then. Characterization tests help ensure that the code’s behavior stays consistent after you’ve made your new changes.

Once you’ve figured out what to test, you need to think about how to get started.

Getting Started

HIGH-LEVEL AND LOW-LEVEL TESTS

For this article, I’ll use the term “high level” to describe tests that exercise top-level classes (e.g., Web framework actions) or public APIs, and “low level” for tests that utilize the individual objects or components (e.g., a sales tax calculator) of your system in isolation. The effectiveness of these test types can be measured in terms of *depth*—how many different components are exercised—and *breadth*—the number of different paths or data combinations executed by the test.

High-level tests provide deep depth and narrow breadth: deep depth because they exercise many layers of the system together, but narrow breadth because they normally exercise just a few paths through the code. But this depth comes at a cost. You need to manage a lot of dependencies and setup before each test. Furthermore, high-level test failures may be hard to diagnose—you might get feedback that a record didn’t appear in the database, but you won’t easily know if it was because of bad input data, a database problem, or some logic error in any of the collaborating objects.

If you have no tests at all, high-level tests that use a real database exercise a lot of the system and can provide confidence that the system is wired together correctly. These tests usually start just below the user interface by accessing the Web framework actions or services directly. If at all possible, avoid testing directly through the GUI since the user

interface typically changes frequently and leads to a lot of incorrect tests and test maintenance.

Low-level tests are the opposite: shallow depth and wide breadth. Low-level tests are easier to set up with a variety of scenarios since you are working with objects in isolation and can more directly specify desired test setup behavior, such as creating test-only objects that always throw exceptions. This makes low-level tests better for isolating behavior and diagnosing failures. However, these tests typically exercise one layer of your application (e.g., business objects or services) and provide shallow depth.

Ultimately, a well-tested system has a combination of both test types: high level to ensure the system is wired correctly and low level to ensure all the business cases are covered.

For most developers, focused low-level tests are the best way to get started. They're quick to write and run, which means they're more likely to be run frequently, and that means more good feedback and a lot of little successes. Taken on their own, these little tests may seem trivial, but put them all together and you've got the beginnings of a regression suite. Build experience writing tests while learning about the properties of good tests. The Pragmatic Programmers publish an excellent book—*Pragmatic Unit Testing*—with editions for Java/JUnit and C#/NUnit, which does a great job teaching the fundamentals.

After you're comfortable writing low-level tests, move on to high-level tests. If you use a database or external systems as part of these tests, you'll need to ensure these are set up in a repeatable fashion before every test. This may require work other than coding, such as creating your own database schema that you can change as part of your tests, figuring out how to start up your own local copy of a dependent server, or creating mock versions of databases or servers.

Once you've gained experience writing both high-level and low-level tests, you'll learn that starting at the lowest level of your application and moving your way up to high-level tests

isn't always the right strategy. Writing low-level tests offers quick rewards, but it can be exhausting to climb level after level of your application. Sometimes it's a good strategy to start with a high-level test to ensure the feature is working end to end, then flesh out the details with low-level tests as necessary.

I faced a similar scenario with a developer. We started writing tests for his new feature at the lowest level of the application. We wrote tests, moving up one layer at a time. After a morning of doing this, we were both exhausted and hadn't yet written a high-level test to verify that the feature worked! In retrospect, it would have been better to start with a high-level test; that way we would have had time to focus on other scenarios to

```
public void testSave() {  
    action = // ... setup code omitted  
    action.setFullName("Nigel Tufnel");  
    action.execute();  
}
```

Listing 1: A bad test

```
public void testSaveShouldCreatePerson() {  
    action = // ... setup code omitted  
    action.setFullName("Nigel Tufnel");  
    String result = action.execute();  
  
    assertEquals(SUCCESS, result);  
    assertPersonCreatedWithName("Nigel Tufnel");  
}
```

Listing 2: A better test

test or refactor the code while the task was still fresh in our minds.

There's a special kind of high-level test I briefly mentioned earlier that controls the application through the user interface. For a Web application, this means automating a Web browser with a tool like Selenium or Watir (see the StickyNotes for links to tools). For a desktop application, this means using libraries like Abbot or White. If you're just learning how to write tests, starting with automated GUI tests is almost always a mistake. They're difficult to reproduce because it's hard to set up the data, they're slow to execute, and they're the most brittle and costly tests. However, these tools are great for smoke tests used to ensure your application installs

and operates correctly end to end. They are not a replacement for a curious—and perhaps devious—tester who can use the application in ways that were never anticipated. For example, it's easy to write a test that fills in invalid form data in a Web page, but a tester might try to click the back button, open a separate copy of the page in another window, and access your application in two windows simultaneously. Instead, start out by automating the simple, repeatable cases just beneath the GUI layer.

Common Problems

TESTS AREN'T CATCHING REGRESSION BUGS!

Good tests catch bugs; bad tests let them slip by undetected. To ensure your tests are good, make some devious changes in your production code and ensure the tests fail in the way you expect. Invert some boolean conditional tests, do one less iteration in a loop, swap some assignment operators with equality checks (`=` for `==`), don't actually save a record to the database, or include whatever sort of mistake might make sense. Do your tests catch the error? If not, evaluate whether you're missing a test or if one of the tests is missing some key detail.

For example, imagine you're writing a test for a customer management Web application. Listing 1 shows an example of a bad test for a "Create a Person" action—the test never checks to see if the person was actually created. In fact, this test will fail only if the code throws an exception. Listing 2 shows a better test, which verifies the action's result code and ensures that a person was created. Note also that we've created a custom assertion method to make the test easier to read. This method could check a real database or a faster in-memory version specifically used for tests.

I CAN'T CHANGE THE DATABASE SINCE IT AFFECTS THE REST OF THE COMPANY

If you are writing tests that use the database, it's much easier to have your own private instance that you can set up and tear down at will. This also means


```

public boolean addAll(int index, Collection c) {
    if(c.isEmpty()) {
        return false;
    } else if( size == index || size == 0) {
        return addAll(c);
    } else {
        Listable succ = getListableAt(index);
        Listable pred = (null == succ) ? null : succ.prev();
        Iterator it = c.iterator();
        while(it.hasNext()) {
            pred = insertListable(pred,succ,it.next());
        }
        return true;
    }
}

```

Figure 1: Sample code coverage annotated by EclEmma

that the build machine should have its own instance that it can use, just like each developer. If it's hard to get the database schema set up reliably in an automated fashion, consider versioning your database changes. Add a version table to keep track of the current schema version, and then create all your changes in incremental SQL scripts that update the schema version after they run. This allows you automatically to re-create or update any database, whether it's the local copy on your workstation or in production. Ruby on Rails uses this technique, as do tools like dbdeploy and migratordotnet.

If you can't get your own instance of the database, there are some techniques you can use to help isolate yourself from the rest of the company (but fight tooth and nail for your own private instance or schema, otherwise your build might occasionally fail because someone else botched the schema). In his book *xUnit Test Patterns*, Gerard Meszaros describes several options for using reusable database fixtures. The premise is that your tests either insert unique data on every run or depend on some data that is always expected to be present. One pattern starts a transaction at the beginning of the test, exercises the system, and then rolls back at the end of the test, preventing any changes from persisting and altering the test database.

HOW CAN MY TEAM MEASURE ITS PROGRESS?

Code coverage is a measure of how much production code your automated

tests exercise, stated as a percentage (for example, "65 percent of my production code is exercised by tests"). While code coverage alone is not a useful metric to determine how safe your application is from accidental regressions (there might be tests, but they might be bad tests), it is a great motivator and teaching aid. Since you're starting out, your coverage number will be low, so set a goal to always have code coverage increase rather than targeting an arbitrary percentage.

Code coverage tools like EclEmma can display production code coverage in your IDE by coloring tested lines of code green and untested lines red. There is no underestimating the "wow" factor of seeing code you've written turn green or red, and it provides great immediate feedback. See Figure 1 for an example.

IT'S HARD TO TEST MY OBJECTS IN ISOLATION

One of the most difficult aspects of starting testing can be getting your code into a test harness—a repeatable configuration of tests with several different scenarios. If your objects under test talk directly to other difficult-to-control components, use dependency injection (see the StickyNotes for a link) to pass in your own test-specific versions. For example, to avoid sending emails to your operations staff every time you run a unit test that verifies a system outage procedure, pass in a fake mail server connection and ensure that a message gets sent via the fake server. Your test will be much more repeatable, and you won't get nasty letters from operations.

By writing more tests, you'll start to learn how to write production code that is easier to test in isolation. However, it won't happen overnight, and it probably will mean making some changes to your existing code to expose testing hooks. Think of these as small steps on the way to a better design. If you have a class that is hard to test, try making a testing-specific subclass that overrides the problematic behavior. It may feel strange to change your production code in awkward ways just to make testing easier, but if you can use it to write automated tests that provide a safety net, you may be able to change your design to remove that test-specific class entirely. Again, see Feathers's excellent *Working Effectively with Legacy Code* for a wealth of techniques for wrangling code into a more testable state.

One Step at a Time...

Just as writing clean code takes discipline, so, too, does writing good automated tests. It's a rewarding practice, but be prepared: It's going to be difficult at first. Start out slow—small, focused low-level tests. Set realistic goals for yourself, and celebrate your successes, such as your first test that runs in the automated build, the first time a test catches a regression bug, or the first test that uses the database. Along the way, verify that your tests are really delivering value—break the production code and ensure a test catches the error.

And, when you become comfortable with writing tests last, I recommend at least trying to write your tests before the production code. I personally find the process of working in small steps with frequent feedback a rewarding and energizing experience. But, some developers love it and some hate it, so decide for yourself. Rather than debating the merits of Test First versus Test Last, let's all celebrate that a good test suite provides us confidence that our code is doing what we expect, and that means we write code more confidently and sleep easier at night.

That sounds like a good life to me.

{end}

SCRUM SUCKS!

But failed projects suck more...
so stop failing!

NOW is the time to improve your
organization, your team, your process, yourself



Scrum Training
Agile Team/Project Rescue
Visioning and Goal Setting
Leadership and Team
Training/Coaching
Organizational Culture Change
Agile Work Environments

Contact Bob Schatz,
bobschatz@agileinfusion.com,
215-435-3240



Why SmarteSoft?



- ✓ Easiest to Learn & Use
 - ✓ 5x Faster to Implement
 - ✓ Outstanding Support
- Any questions?

Test Automation : Test Deployment : Test Management : Load Test

sales@smartersoft.com
(512) 782 9400
www.smartersoft.com

SmarteSoft™
Test » Automation » » » Made Easy

Q/P's World-Class Tools, Services and Data can Pave your Road to Success

Measurement & Estimating Tools
PQMPlus™
Software Measurement &
Reporting (SMR)

CONSULTING SERVICES
-MEASUREMENT
-ASSESSMENT
-PROCESS IMPROVEMENT

SOFTWARE
BENCHMARK DATABASE

World 's Largest Functional
Size-Based Database

Lower Costs
Better Quality
Shorter Schedules
Accurate Estimates
Improved Productivity



Q/P MANAGEMENT
GROUP, INC.



Advantage!



SEI Partner

For more information:

www.qpmg.com

moreinfo@qpmg.com

contact our office at 781.438.2692

1 Things You Might Not Know About

Test Process Improvement

by Rick Craig

- 1 **TEST PROCESS IMPROVEMENT NEVER ENDS.** Organizations are rarely static, so test managers and process engineers must treat process improvement as an ongoing activity. Intermediate milestones should be established and may subsequently be met, but the long-term success of the process improvement initiative hinges on continuously striving for advancement and refinement.
- 2 **USE A MODEL OR REFERENCE POINT TO BASELINE CURRENT PRACTICES AND IDENTIFY LIKELY CHANGE AREAS.** Of all the good process reference models available today, I prefer to use the Test Process Improvement Model (TPI). The TPI model is easy to use and accomplishes two important goals: 1) It baselines current practices, and 2) it identifies and prioritizes likely process improvement areas. It's also a good idea to baseline traditional measures of test effectiveness such as DDP and coverage at the beginning of the process improvement initiative in order to measure the success of the changes.
- 3 **CREATE A PLAN OR STRATEGY FOR THE PROCESS IMPROVEMENT ENDEAVOR.** A normal project or test plan template can be used as a starting point since it contains much of what we need for our process improvement plan: responsibilities, training, pass/fail criteria, schedule, risks, priorities, etc.
- 4 **MORE IS NOT ALWAYS BETTER.** Sometimes process improvement means reducing processes or documentation rather than increasing it. Make sure that the new or changed processes are flexible enough to handle projects of different types and scope. A process that is too prescriptive is wasteful and demoralizing.
- 5 **USE INFLUENCE LEADERS OR CHANGE AGENTS TO FACILITATE THE PROCESS IMPROVEMENT.** Influence leaders are members of the team who are perceived by their peers as leaders due to their experience, personality, training, communication skills, etc. They may not necessarily be "managers" in the organizational sense. Ideally participants should be enthusiastic volunteers who see the need for change. Influence leaders also facilitate buy-in.
- 6 **BUY-IN IS ESSENTIAL.** If the process improvement initiative is to successfully implement real change, the stakeholders and practitioners must feel that they are part of the process. Training, feedback loops, and participation are key to buy-in.
- 7 **INTRODUCE CHANGE INCREMENTALLY.** Trying to implement many different programs simultaneously is difficult to manage and frustrating for the participants. If multiple changes are introduced at the same time, it is difficult to gauge the effectiveness of each change. When possible, pilot each change with a willing team. Hint: If multiple process improvement initiatives are ongoing simultaneously (including development), make an effort to consolidate or at least coordinate these efforts.
- 8 **FIND A SPONSOR.** Bottom-up or grass-roots change is possible but is typically difficult and costly. Having a senior management advocate for the process improvement initiative can facilitate budgeting, staffing, training, and buy-in from other senior managers.
- 9 **IDENTIFY AND IMPLEMENT PROCESS IMPROVEMENTS AT THE TEAM LEVEL.** Process improvement can (and should) be conducted at the organizational level but should not preclude individual teams from pursuing the changes that benefit them most. Most process models (including TPI) adopt the lowest common denominator for each identified key area. If the process improvement assessment and subsequent implementation are conducted at too high a level, some teams will be waiting on the other teams to "catch up" before they can implement meaningful change.
- 10 **MANAGE EXPECTATIONS.** Faster, better, and cheaper are all worthy high-level goals for process improvement, but it is frequently impossible and even unproductive to try to achieve all three at the same time. The process improvement initiative will have a greater chance of success if one of these three goals is chosen as the primary goal. The goal chosen may also have an impact on which processes are chosen for improvement.

Putting the Kart before the Horse?

by Antony Marcano

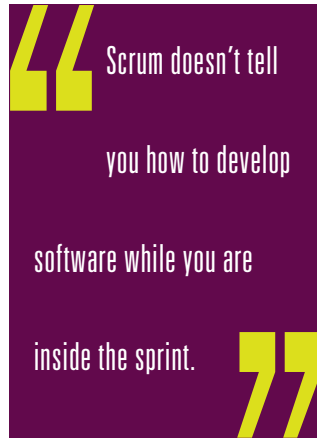
Deep down, I'm a frustrated racing driver—one who occasionally indulged his passion at his local go-kart circuit (coincidentally the same track on which Lewis Hamilton, current Formula One world champion, first learned to race). Later, I raced in an adults' amateur karting league—pure exhilaration! Super-car acceleration, 70 mph+ top speeds, and no seatbelts!

My passion for racing seems to have rubbed off on my ten-year-old son. After his many polite and persistent requests, I recently took him karting for the first time.

While experienced racing drivers make speed seem effortless, consistently fast lap times are physically demanding and require enormous skill—even in go-karts. At the open practice session my son and I attended, novice drivers seemed to approach their first lap as if it really were as easy as experienced drivers make it look. They floored the throttle and flew down the straight only to crash—painfully and expensively—into the barrier at the first hairpin turn.

Thanks to a little coaching from me, my son started out slowly. At low speed he practiced the line I showed him on each corner, gradually improving the quality of his cornering and then increasing his speed. He wasn't concerned with going full-throttle. He was developing the cornering skills needed to achieve consistently fast average speeds. I suggested an earlier braking point here and a later apex there and by the end of the afternoon he was achieving the fastest and most *consistent* lap times—even faster than the adults (except for me, of course).

Organizations starting their transition to agile methods with Scrum are much like the novice drivers trying to go full-throttle before learning how to take the corners. Before being able to truly bend



and change their software with speed and ease, they fly into short iterations and frequent incremental product delivery—crashing painfully into the barrier of hard-to-change software.

As described on controlchaos.com, Scrum is “... used to manage and control complex software and product development using iterative, incremental practices.”

Using timeboxes of two to four weeks, called “sprints,” Scrum arms you with techniques and memorable terminology for *managing* iterative and incremental development. At the end of each sprint, working software should be demonstrated for customer feedback or immediate release. Other than techniques for planning, managing, and sharing information, Scrum doesn't tell you how to develop software while you are inside the sprint. For greatest success, it relies on your team's having the software practices and associated skills for creating software that is truly “soft,” malleable, and inexpensive to change.

Another racing analogy comes to mind to explain this. Scrum is more akin to what happens trackside—how the racing team plans the race, monitors lap times, communicates information, and reflects and learns from the race event. It doesn't tell you how to build a race car or how to drive it. It relies on the race engineers and the driver to have the skills to do that already.

Despite this, many organizations declare that they are “going agile” and see Scrum as the panacea to the problems standing between them and that goal. They start with the expectation that the light at the end of the agile-adoption tunnel is their mastery of Scrum, only to find that it is the spotlight that illuminates the brittle nature of a product created using legacy development practices.

In my experience, these organizations:

- Assume that, once implemented, the software will be expensive to change, so they create speculative designs intended to solve future problems—making the design far more complex than necessary
- Develop the software in a way that makes evolutionary design impractical, causing them to slice the product increments by “horizontal” architectural component, which makes it difficult to demonstrate the value to the customer, reducing trust and delaying feedback
- Believe that writing unit tests slows you down, leading to low unit test coverage and infrequent test runs and causing deteriorating throughput by deferring the detection of bugs, often until it is too late
- Underestimate the gradient of the learning curve in automating acceptance tests, abandoning it before the true value can be realized, which reduces clarity of requirements and consuming the sprint with increasing manual regression testing overheads
- Continue with a phased development cycle separating programmers and testers (and other disciplines), making communicating changes in requirements unnecessarily difficult and causing friction between disciplines

After an eventual peak in demonstrable features, these behaviors contribute to an ever-decreasing velocity and an increasing proportion of each new sprint spent debugging and testing bug fixes. The term “agile” is quickly replaced with “fragile” and feels more like *brittleware development* than software development.

The team reacts by spilling activities, like testing, into later sprints or abandoning testing altogether and pulling

design into earlier sprints. Superficially, these teams appear to be agile with index cards and project boards and daily stand ups called scrums. But the time between successful demonstrations of working software continually grows, well beyond a single sprint. Agile this is not.

Now, I do appreciate that there are many Scrum-first agile adopters out there who are experiencing success. However, this perception of success may well be relative to how the organization was doing before using Scrum and very much related to what they expected to happen once they'd mastered Scrum.

Adopting Scrum *before* developing the technical practices required to create easy-to-change software does have a benefit. At the end of each sprint that fails to show a product increment or refinement of value to the customer, Scrum painfully highlights how hard it really is to change the software produced by these legacy practices. Sometimes this is the only way to influence cultural change—for the organization to learn a few lessons the hard way.

That was how the lesson was learned

for one delegate who spoke up during a presentation at a conference I attended last year. With the benefit of hindsight, he wished that his organization had started with the agile technical practices first. Strangely, it was only after hearing his story that I reflected on my own experience in that regard. It was only then that I *really* appreciated how much “going agile” with Scrum first was actually putting the cart before the horse.

Interestingly, as I write this column, I realize that I'd already been doing exactly what that delegate had wished for. Over the past year, one of my clients has, with my help, gradually reduced the time between delivering releasable software from three months to two weeks. The client did this by first focusing on its technical practices. Much like my son's approach to delivering faster lap times, the client started slow and gradually developed the skills required to deliver software more quickly, frequently, and consistently through improved quality.

It was only when the frequency of each releasable version of its product had been

reduced to about six weeks—coincidentally closer to the typical recommended length of a sprint—that the team started to employ any noteworthy techniques from Scrum. I guess the team's skill on the track had reached a point where it saw the value of having better trackside support and information. I suspect that for my son, at the rate his lap times are improving, it'll not be long before he will want the same.

{end}

HAVE THE LAST WORD!

If you have a point to make or a side to take on issues and trends that affect the industry, we want to hear from you.

We are looking for insightful, thought-provoking commentary for possible use as **The Last Word**.

Please send an abstract to
editors@bettersoftware.com.

Index to Advertisers

Agile Infusion	www.agileinfusion.com	37
Better Software Conference & EXPO 2009	www.sqe.com/BetterSoftwareConf	7
CHARISMATEK	www.charismatek.com	Inside Back Cover
Cognizant	www.cognizant.com	2
Hewlett-Packard	www.hp.com/go/software	Back Cover
McCabe Software, Inc.	www.mccabe.com/bettersoftware	8
Phantom Automated Solutions	www.phantomtest.com	17
Q/P Management Group	www.qpmg.com	37
Rally Software	www.rallydev.com/bsm	Inside Front Cover
Seapine	www.seapine.com	1
SmarteSoft	www.smartesoftware.com	37
SQE Training—Agile	www.sqetraining.com/Agile	11
SQE Training—Testing Certification	www.sqetraining.com/Certification	18
STARWEST 2009	www.sqe.com/STARWEST	13
TechExcel	www.techexcel.com	5
ThoughtWorks	www.thoughtworks.com	15
VersionOne	www.versionone.com	25

Display Advertising advertisingsales@sqe.com

All Other Inquiries info@bettersoftware.com

Better Software (USPS: 019-578, ISSN: 1553-1929) is published seven times per year January/February, March, April, May/June, July/August, September/October, November/December. Subscription rate is US \$39.99 per year. A US \$35 shipping charge is incurred for all non-US addresses. Payments to Software Quality Engineering must be made in US funds drawn from a US bank. For more information, contact info@bettersoftware.com or call 800.450.7854. Back issues may be purchased for \$15 per issue (plus shipping). Volume discounts available. Entire contents © 2009 by Software Quality Engineering (330 Corporate Way, Suite 300, Orange Park, FL 32073), unless otherwise noted on specific articles. The opinions expressed within the articles and contents herein do not necessarily express those of the publisher (Software Quality Engineering). All rights reserved. No material in this publication may be reproduced in any form without permission. Reprints of individual articles available. Call for details. Periodicals Postage paid in Orange Park, FL, and other mailing offices. POSTMASTER: Send address changes to Better Software, 330 Corporate Way, Suite 300, Orange Park, FL 32073, info@bettersoftware.com.



Software Project Forecasting - are you driving blind?

Know where you are going, how far it is and how fast you can go BEFORE you start the journey.

Function Point Workbench Release 7

Putting you in the driver's seat for software delivery.

Ask us how the Function Point WORKBENCH can assist you to apply Function Point Analysis and further metrics to help you define, scope, estimate AND MANAGE TO COMPLETION.

USA Resellers include:

www.qpmg.com

www.qualityplustech.com

www.davidconsultinggroup.com

www.softwarems.com

www.softwaremeasurementexpertise.com

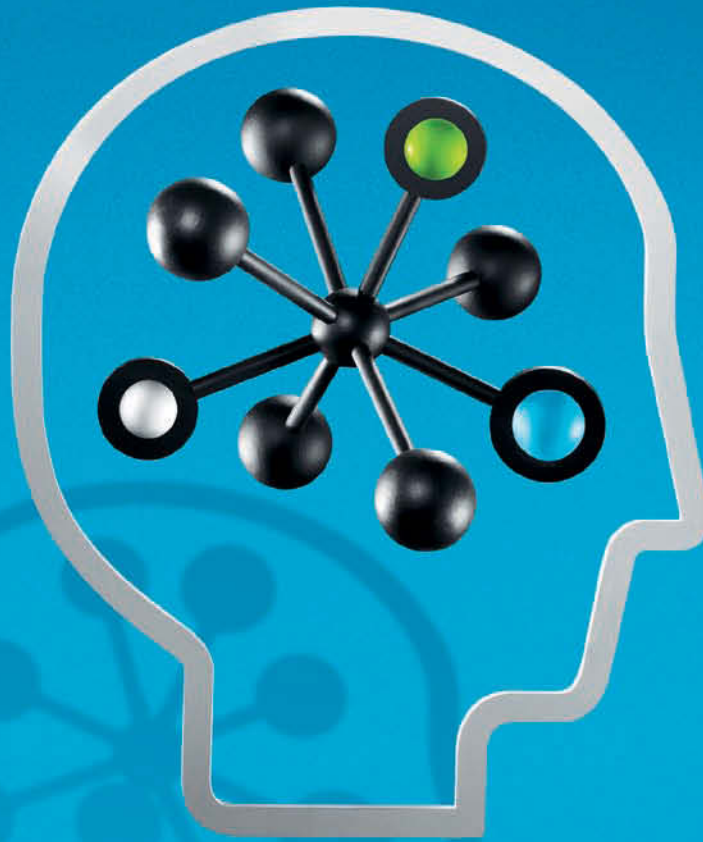
www.spr.com

Email us at info@charismatek.com for more contact details or product information

CHARISMATEK Software Metrics - www.charismatek.com
Aligning the Software Development Process with Your Business Imperatives



CHARISMATEK
SOFTWARE METRICS



ALTERNATIVE THINKING ABOUT APPLICATION LIFECYCLE MANAGEMENT:

Computers Don't Run Your Apps. People Do.

Alternative thinking is looking beyond the development cycle and focusing on customer satisfaction. Because the real application lifecycle involves real people – and the customer's perception is all that matters in the end.

HP helps you see the big picture and manage the application lifecycle. From the moment it starts – from a business goal, to requirements, to development and quality management – (and here is the difference) – all the way through to operations where the application touches your customers.

HP ALM offerings help you ensure that your applications not only function properly, but perform under heavy load and are secure from hackers. (Can't you just hear your customers cheer now?)

Technology for better business outcomes. hp.com/go/alm

